Markt&Technik

Florian Matthes

Pascal mit dem C64

Compiler-Beschreibung · Pascal-Kurs
Tips und Tricks für
Fortgeschrittene

Enthalten:
5¼"-Diskette mit
professionellem
Pascal-Compiler

Pascal mit dem C64

Florian Matthes

Pascal mit dem C64

- Compiler-Beschreibung
- Pascal-Kurs
- Tips und Tricks für Fortgeschrittene

Markt & Technik Verlag AG

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Matthes, Florian:
Pascal mit dem C64 : Compiler-Beschreibung, Pascal-Kurs,
Tips u. Tricks für Fortgeschrittene / Florian Matthes. –
Haar bei München : Markt-und-Technik-Verlag, 1986.
ISBN 3-89090-222-7

Die Informationen im vorliegenden Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können
für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine
Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Buch gezeigten Modelle und Arbeiten ist nicht zulässig.

»Commodore 64« ist eine Produktbezeichnung der Commodore Büromaschinen GmbH, Frankfurt,
die ebenso wie der Name »Commodore« Schutzrecht genießt.
Der Gebrauch bzw. die Verwendung bedarf der Erlaubnis der Schutzrechtsinhaberin.

15 14 13 12 11 10 9 8 7 6 5
89 88 87

ISBN 3-89090-222-7

© 1986 by Markt&Technik Verlag Aktiengesellschaft,
Hans-Pinsel-Straße 2, D-8013 Haar bei München/West-Germany
Alle Rechte vorbehalten
Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Schoder, Gersthofen
Printed in Germany

Inhaltsverzeichnis

	Vorwort	9
1	**Die Werkzeuge**	11
1.1	Warum Pascal?	11
1.2	Was macht ein Compiler?	12
1.3	Das Pascal-System	14
1.3.1	Systemstart	14
1.3.2	Der Editor	15
1.3.3	Der Compiler	17
1.3.4	Das Laufzeitsystem	17
2	**Einführung in Pascal**	21
2.1	Symbole und Syntax-Diagramme	21
2.2	Programmstruktur	25
2.3	Deklaration von Variablen	27
2.4	Anweisungen und Ausdrücke	28
2.5	Einfache Ein- und Ausgabe	30
2.5.1	WRITE	30
2.5.2	READ	32
2.6	Elementare Datentypen	35
2.6.1	Der Typ INTEGER	36
2.6.2	Der Typ REAL	37
2.6.3	Gegenüberstellung REAL und INTEGER	39
2.6.4	Der Typ CHAR	40
2.6.5	Der Typ BOOLEAN	42
2.7	Deklaration von Konstanten	44
2.8	Kontrollstrukturen	45
2.8.1	Anweisungsfolgen	46
2.8.2	Bedingte Anweisungen	47

2.8.3	Fallunterscheidung	50
2.8.4	While-Anweisung	52
2.8.5	Repeat-Anweisung	54
2.8.6	For-Anweisung	55
2.8.7	Sprunganweisung	57
2.9	Die Datenstruktur Array	59
2.9.1	Eindimensionale Arrays	60
2.9.2	Strings	65
2.9.3	Mehrdimensionale Arrays	66
2.10	Deklaration von Typen	71
2.11	Prozeduren	73
2.11.1	Lokalität von Bezeichnern	75
2.11.2	Parameter	78
2.11.3	Funktionen	81
2.11.4	Standardprozeduren	82
2.11.5	Rekursion	82
2.12	Skalare Typen und ihre Operationen	92
2.12.1	Aufzählungstypen	92
2.12.2	Unterbereichstypen	94
2.13	Mengentypen	96
2.14	Der Datentyp Record	99
2.15	Variante Records	102
2.16	Der Datentyp File	107
2.16.1	Sequentiell schreiben	108
2.16.2	Sequentiell lesen	108
2.17	Textfiles	115
2.18	Dynamische Datenstrukturen	121
2.18.1	Lineare Strukturen (Listen)	125
2.18.2	Bäume	132
3	**Tips und Tricks**	139
3.1	Nützliche Pascal-Routinen	139
3.2	Tips zum Editor	148
4	**Dokumentation Pascal-System**	151
4.1	Das Pascal-Menü	152
4.2	Der Editor	153
4.2.1	Allgemeines	153
4.2.2	Gliederung des Bildschirms	154
4.2.3	Cursorsteuerung	156
4.2.4	Primary-Commands	158
4.2.5	Line-Commands	159
4.2.6	Textmodus	161

4.2.7	FIND	162
4.2.8	CHANGE	163
4.2.9	Die Tasten f2 und f4	164
4.2.10	INPUT	165
4.2.11	OUTPUT	166
4.2.12	COPY	168
4.2.13	Fehlermeldungen im Editor	170
4.3	Bedienung des Compilers	172
4.3.1	Wahl der Optionen	172
4.3.2	Meldungen im Compiler	173
4.3.3	Spezielles	174
4.3.4	Rückkehr zu BASIC	176
4.4	Sprachbeschreibung Pascal 1.4	177
4.4.1	Grundsätzliches	177
4.4.2	Sprachumfang	178
4.4.3	Reservierte Wortsymbole	182
4.4.4	Vordefinierte Bezeichner	183
4.4.4.1	Konstantenbezeichner	183
4.4.4.2	Typbezeichner	183
4.4.4.3	Variablenbezeichner	183
4.4.4.4	Prozeduren für dynamische Objekte	183
4.4.4.5	Ein- und Ausgabe	184
4.4.4.6	Arithmetische Funktionen	188
4.4.4.7	Verschiedenes	189
4.4.5	Files in Pascal 1.4	190
4.4.5.1	Übernahme von Programmen mit Files	192
4.4.6	Aktive Kommentare	193
4.4.6.1	Bereichstests	193
4.4.6.2	Include-Files	194

Anhang

A	Syntax-Diagramme Pascal 1.4	195
B	Fehlernummern Pascal 1.4	201
C	Laufzeitfehler	205
D	Operatoren in Pascal	207
E	Literaturhinweise	211
F	Index	213

Vorwort

Das vorliegende Buch richtet sich an Schüler, Studenten und Hobbyprogrammierer, die einen C 64 besitzen und einen praktischen Einstieg in Pascal finden wollen. Dabei werden nur minimale Kenntnisse in der Programmierung vorausgesetzt.

Durch die Einheit Buch-Diskette können die nicht zu unterschätzenden Probleme eines Anfängers bei der Benutzung eines Compilers ausgeräumt werden. Das erste Kapitel beschreibt deshalb zunächst an einem Beispiel Schritt für Schritt die Bedienung des Systems.

Kapitel 2 stellt einen vollständigen Pascal-Kurs für Anfänger dar. Da der beiliegende Compiler den vollen Standard (1) akzeptiert, werden alle Elemente von Pascal vorgestellt. Der Leser soll möglichst früh die Grundlagen von Pascal erlernen, mit denen er erste einfache Programme erstellen kann.

Beispiele sollen nicht Selbstzweck sein, sondern später zumindest als Schema für eigene Problemlösungen dienen. Im gesamten zweiten Kapitel werden Anregungen gegeben, das erworbene Wissen durch eigene *Experimente* am C 64 zu festigen.

Kapitel 3 richtet sich an den fortgeschrittenen Anwender. Während die Einführung auf systemspezifische Programme verzichtet (Sprites, Grafik, Sound), werden hier Tricks und Tips zum Pascal-System gegeben.

In Kapitel 4 ist die Dokumentation des Pascal-Systems zusammengestellt. Sie gibt klare Auskunft auch über Details des Editors und Compilers.

1 Die Werkzeuge

1.1 Warum Pascal?

An dieser Stelle sollen nicht weitschweifig die grundsätzlichen Vorteile der Strukturierten Programmierung dargestellt werden, sondern die sinnvollen Anwendungsgebiete für Pascal auf dem C 64, einem typischen Homecomputer, gezeigt werden.

Einerseits kann man die Sprache Pascal um ihrer selbst willen benutzen: Man arbeitet mit Pascal, um eine moderne Programmiersprache zu beherrschen und vielleicht das Wissen in Schule, Universität oder Beruf zu verwenden.

Andererseits bieten einige Hobbyanwendungen (Logikspiele, Dateiprogramme, Mathematikprogramme) Beispiele für Gebiete, in denen eine Sprache mit mächtigeren Strukturen für Daten und Programme deutliche Vorteile gegenüber BASIC besitzt.

Schließlich sprechen auch die höhere Ausführungsgeschwindigkeit und der kompakte Code bei großen Programmen für die Verwendung von Pascal.

Nicht zu vergessen ist die Portabilität der Programme. Ein Programm, das auf dem C 64 in Pascal erstellt wurde und keine speziellen Eigenschaften des C 64 benutzt (SYS-Befehle etc.), kann direkt auf einen IBM-PC, ATARI 520 ST oder gar einen Großrechner an der Universität übernommen werden.

Neben diesen Vorteilen dürfen aber auch die Grenzen von Pascal nicht vergessen werden. Ein schnelles Action-Spiel wird man besser mit einem Assembler erstellen, und Programme mit intensiven String-Operationen sind immer noch einfacher in BASIC zu formulieren. Sicherlich wird aber die Erfahrung mit dem strikten Formalismus in Pascal auch den Programmierstil in diesen Sprachen verändern.

1.2 Was macht ein Compiler?

Um die Funktionsweise des Pascal-Systems zu verstehen, muß zunächst die Aufgabe eines Compilers erläutert werden.

Vielleicht haben Sie schon gehört, daß kein Mikrocomputer direkt BASIC oder Pascal *versteht*, sondern nur in seiner speziellen Maschinensprache programmiert werden kann. Andererseits können Sie ja offensichtlich den C 64 mit Befehlen wie PRINT 6*4 oder GOTO 9 zu sinnvollen Tätigkeiten bewegen. Darüber hinaus verspricht Ihnen dieses Buch, auch in Pascal mit dem C 64 kommunizieren zu können.

Die Lösung dieses Dilemmas ist die Existenz von Hilfsprogrammen, die BASIC oder Pascal in die primitive Maschinensprache übersetzen.

Diese Hilfsprogramme selbst sind vollständig in der Maschinensprache des Mikroprozessors (des 6510 beim C 64) geschrieben und somit von diesem direkt ausführbar.

Beim C 64 befindet sich dieses Hilfsprogramm für BASIC bereits beim Einschalten im Rechner, da es zusammen mit dem Betriebssystem unlöschbar in sogenannten ROMs gespeichert ist. Wenn Sie in BASIC eine Zeile mit Zeilennummer eingeben, so wird diese Zeile im Rechner gespeichert. Beim Programmstart mit RUN wird das Programm Befehl für Befehl gelesen. Für jeden Befehl wird ein kleines Programm in Maschinensprache aufgerufen, das den jeweiligen Befehl ausführt. Bei dem Befehl PRINT 6*3 würde z.B. eine Multiplikationsroutine und dann eine Aus-

gaberoutine gestartet. Falls Sie bei der Eingabe Fehler gemacht haben, meldet dies das System mit Angabe der Zeilennummer des fehlerhaften Befehls:

```
SYNTAX ERROR IN 312
```

Ein Hilfsprogramm, das die Ausführung eines Programmes nach diesem Schema schrittweise organisiert, nennt man **Interpreter**. Durch diese interpretative Ausführung können Sie in BASIC beliebig zwischen Programmausführung und Programmänderung wechseln und sogar Befehle ohne Zeilennummer direkt ausführen.

Das Pascal-System auf der beiliegenden Diskette enthält einen **Compiler**. Dies ist ein Programm, das ebenfalls eine Übersetzung der *höheren* (problemorientierten) Programmiersprache Pascal in Maschinensprache vornimmt. Der Übersetzungsvorgang unterscheidet sich wie folgt von der Arbeitsweise eines Interpreters:

Zunächst erstellen Sie ein komplettes (!) Programm in Pascal. Dieses Programm geben Sie mit einem Editor, also einem Textverarbeitungsprogramm, ein. Dieses Programm heißt **Quelltext** (source code). Der Compiler liest diesen Programmtext in einem Durchlauf. Dabei prüft er, ob das Programm den Syntax-Regeln für Pascal entspricht. Eventuelle Fehler werden mit einem Hinweis auf die Art des Fehlers markiert. Gleichzeitig werden fehlerfreie Anweisungen in eine Folge von Befehlen in Maschinensprache übersetzt.

Ergebnis der Übersetzung ist also ein Programm, das vom Rechner ohne weitere Hilfsprogramme ausgeführt werden kann. Dieses Programm bezeichnet man als **Objektprogramm** (object code). Theoretisch könnten Sie jetzt den Quelltext löschen, da dieser nicht mehr benötigt wird. Natürlich werden Sie das nicht tun, da das Programm noch logische Fehler enthalten kann, die der Compiler nicht entdeckt.

Zur Korrektur von logischen oder syntaktischen Fehlern müssen Sie wieder von vorn anfangen: Der Quelltext muß nach einer Korrektur neu übersetzt werden. Den Vorteil einer vollständigen Prüfung auf syntaktische Korrektheit erkauft man sich also durch einen größeren Übersetzungsaufwand. Ein weiterer Nachteil besteht darin, daß bei Fehlern bei der Ausführung des Objektprogrammes (z.B. Division durch null) kein Verweis auf die Fehlerposition im Quelltext existiert.

1.3 Das Pascal-System

In diesem Kapitel werden noch keine Eigenschaften der Sprache Pascal vorgestellt, sondern nur die ersten Schritte bei der Bedienung des Pascal-Systems genau erklärt. Nachdem Sie dieses Kapitel bearbeitet haben, kennen Sie das Zusammenspiel der Komponenten des Systems, so daß Sie sich ohne Probleme in der Dokumentation in Kapitel 4 zurechtfinden werden.

1.3.1 Systemstart

Entfernen Sie alle Erweiterungsmodule, und schalten Sie den C 64 aus und dann wieder ein, um alle geladenen Hilfsprogramme zu löschen. Mit

```
LOAD "PASCAL-SYSTEM",8
```

laden Sie das System von der beiliegenden Diskette. Auf einer anderen Diskette legen Sie zunächst eine Sicherheitskopie des Programmes an:

```
SAVE "PASCAL-SYSTEM",8
VERIFY "PASCAL-SYSTEM",8
```

Die Beispielprogramme (xxx.P) können Sie so **nicht** kopieren. Dazu benutzen Sie den Editor (s. Abschnitt 1.3.2). Haben Sie das System mit LOAD geladen, so gelangen Sie mit dem Befehl RUN in das zentrale Menü des Systems. In diesem Pascal-Menü können alle Teile des Systems aufgerufen werden. Ab jetzt benötigen Sie die Programmdiskette nicht mehr im Diskettenlaufwerk. Für die Speicherung von Pascal-Quelltexten und Objektprogrammen können Sie jetzt eine andere (formatierte) Diskette einlegen.

Alle Eingaben im System sind so organisiert, daß Sie mit möglichst wenigen Zwischenschritten jede Funktion erreichen können. Dabei besitzen im allgemeinen die Zeichen '*' und '?' eine Sonderfunktion. Eingaben bei blinkendem Cursor müssen mit der RETURN-Taste beendet werden.

Grundsätzlich wird bei längeren Operationen (Laden, Speichern, Compilieren) eine Abfrage der RUN/STOP-Taste vorgenommen, so daß die Ausführung abgebrochen werden kann.

1.3.2 Der Editor

Um Pascal-Quelltexte einzugeben und zu verändern, enthält das System einen komfortablen Full-Screen-Editor. Mit diesem Programm können Sie den Bildschirm wie ein Fenster in allen vier Richtungen über den Text verschieben und direkt Änderungen vornehmen, ohne sich um Zeilennummern zu kümmern.

Sie sollen zur Übung folgenden Quelltext eingeben:

```
PROGRAM PROGRAMM1 (OUTPUT);

  VAR I, N: INTEGER;
      R   : REAL;

BEGIN
  WRITE("N="); READLN(N);
  FOR I:= N TO 2*N DO
    BEGIN
      R:= 1/I;
      WRITELN(I:3,R:15)
    END
END.
```

Listing 1: *Übungsprogramm*

Den Editor erreichen Sie aus dem Pascal-Menü durch die Eingabe eines Namens aus maximal 16 Zeichen. Dieser Name darf die Zeichen '*', '$' und '?' nicht enthalten. Unter diesem Namen speichert der Editor den Text später auf der Diskette.

Als Namen geben Sie an dieser Stelle PROGRAMM1.P ein. Später können Sie dann alle Quelltexte an der Endung '.P' erkennen. Der Editor sucht diesen Quelltext auf der eingelegten Diskette. Da noch kein Text mit diesem Namen existiert, teilt Ihnen der Editor mit, daß er einen neuen Text anlegen wird. Sie müssen nun eine Zahl eingeben, die die maximale Länge einer Textzeile bestimmt:

```
==>60
```

Wenn Sie die Eingabe mit RETURN abschließen, erscheint das eigentliche Editor-Bild (s. Abschnitt 4.2). In der ersten Zeile wird die Nummer der ersten Textspalte auf dem Bildschirm angegeben. Rechts außen in dieser Zeile steht der Betrag, um den das Textfenster beim Blättern (scrollen) verschoben wird. HALF bedeutet jeweils eine halbe Bilschirmseite. Das Blättern selbst erfolgt mit den Funktionstasten:

f1 verschiebt das Fenster nach oben.
f3 verschiebt das Fenster nach unten.
f5 verschiebt das Fenster nach links.
f7 verschiebt das Fenster nach rechts.

Der Cursor wird nicht blinkend dargestellt und wie üblich mit den Cursortasten bewegt. In der obersten Zeile (weiß) werden auch Befehle (Primary-Commands) eingegeben. Geben Sie dort den folgenden Befehl ein:

```
COLUMNS
```

Damit ein Befehl ausgeführt wird, müssen Sie die SHIFT- und die RETURN-Taste drücken. Es erscheint eine Zeile, in der Spaltenmarkierungen eingetragen sind.

Da der Textspeicher leer ist, stehen unterhalb der Spaltenmarkierungen die Zeilen TOP und BOTTOM direkt untereinander. Sie stehen immer vor der ersten und nach der letzten Textzeile. Um nun das Programm einzugeben, erzeugen Sie sich zunächst einige Leerzeilen. Dies geschieht, indem Sie in der Zeile TOP ganz links (vor den Sternen) den Befehl (Line-Command)

```
I10
```

eingeben. Wenn Sie wieder SHIFT-RETURN drücken, werden am Textanfang 10 Leerzeilen eingefügt. Jetzt steht das Fenster am Textende, so daß nur die Zeile BOTTOM sichtbar ist. Mit f1 können Sie an den Anfang der Leerzeilen gehen. Die eigentliche Texteingabe erfolgt rechts von den weißen Zeilennummern. Dabei springt der Cursor bei Betätigung der RETURN-Taste auf den nächsten Zeilenanfang.

Wollen Sie nach einer Zeile eine Leerzeile einfügen, so geben Sie im Zeilennummernbereich 'I' ein. Analog löscht man eine Zeile, indem man bei ihrer Zeilennummer ein 'D' eingibt. Wiederum wird der Befehl mit SHIFT-RETURN ausgeführt. Haben Sie den gesamten Text fertig eingegeben, so tippen Sie in der ersten Bildschirmzeile den Befehl (Primary-Command)

```
END
```

Nach SHIFT-RETURN wird der Text auf der eingelegten Diskette gespeichert. Anschließend erreichen Sie wieder das Pascal-Menü.

1.3.3 Der Compiler

Um den Quelltext im Textspeicher zu übersetzen, geben Sie im Pascal-Menü ein Dollarzeichen ein. Nach dem Drücken der RETURN-Taste meldet sich der Compiler (Pascal 1.4).

Hier soll nicht näher auf die möglichen Optionen (PCODE-START, LISTING TO PRINTER) eingegangen werden. Sie müssen nur die vorgegebenen Eingaben mit RETURN bestätigen. Während der Compilation wird der Quelltext am Bildschirm angezeigt.

Sollten Sie sich bei der Eingabe des Programmes vertippt haben, markiert der Compiler den entsprechenden Fehler mit einem Pfeil. Durch die Eingabe von '*' brechen Sie dann die Übersetzung ab. Drücken Sie nur RETURN, so wird die Compilation fortgesetzt. Am Ende erscheint die Meldung

```
ERRORS DETECTED: xx
PCODE FROM xxxx TO xxxx

(HIT RETURN FOR MENUE)
```

Mit der RETURN-Taste kehren Sie zum Pascal-Menü zurück.

Traten bei der Übersetzung Fehler auf, so müssen Sie vom Menü mit der Eingabe

```
PROGRAMM1.P
```

zum Editor zurückkehren und im Text Korrekturen vornehmen. Nach der Rückkehr zum Pascal-Menü (mit END) können Sie die obigen Schritte wiederholen.

1.3.4 Das Laufzeitsystem

Ist der Text endlich fehlerfrei, so möchten Sie sicher als Lohn Ihrer Arbeit das Programm auch einmal ausführen. Dazu verlassen Sie das Pascal-Menü mit der Eingabe '*'. Es erscheint die übliche Meldung von BASIC:

```
READY.
```

Das Objektprogramm steht jetzt im Speicher. Es kann mit RUN, SAVE und VERIFY wie ein BASIC-Programm gestartet, gespeichert und geprüft werden. Sie dürfen jedoch keine Zeilen löschen oder hinzufügen, da sonst das Pascal-System nicht mehr korrekt arbeitet.

Das Beispielprogramm druckt nach der Eingabe einer ganzen Zahl N alle Zahlen zwischen N und 2*N mit ihrem Kehrwert aus.

Um zu demonstrieren, wie das System auf einen Fehler während der Ausführung reagiert, wählen Sie N=0: Diese Eingabe hat eine Division durch 0 zur Folge. Es erscheint folgende Meldung:

```
DIVISION BY ZERO
 ERROR AT xxxx
```

xxxx ist eine dezimale Adresse im Objektprogramm. Notieren Sie sich diese Zahl. Um nun den fehlerhaften Divisionsbefehl im Quellprogramm zu lokalisieren, muß der Compiler erneut gestartet werden. Deshalb kehren Sie durch die Eingabe eines Sterns ('*') von BASIC zum Pascal-Menü zurück.

Dort starten Sie den Compiler mit '$'. Bei der ersten Eingabe wählen Sie die Option LOCATE ADDRESS. Dazu geben Sie die Zahl xxxx aus der Fehlermeldung anstatt der angezeigten Zahl 1 ein. Die weiteren vorgegebenen Eingaben bestätigen Sie nur mit RETURN. Wiederum wird der Quelltext aufgelistet. Jedoch erscheinen unter dem Divisionsbefehl (R:=1/I) ein Pfeil und die Meldung

```
**** ERROR 0 IN 9
```

Der Pfeil markiert also die Position des Laufzeitfehlers. Alle Fehlernummern sind im Anhang B mit Erklärung aufgelistet.

Bild 1 zeigt noch einmal zusammenfassend die Komponenten des Pascal-Systems.

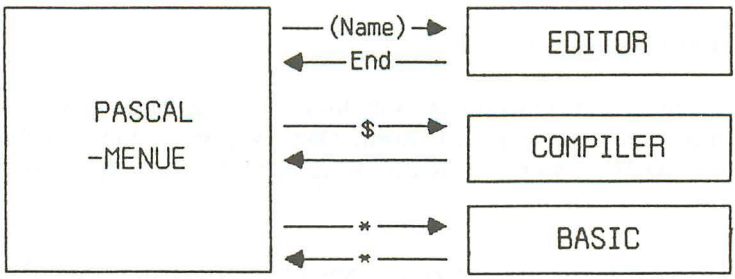


Bild 1: *Systemstruktur*

Zur Übung können Sie jetzt Sicherheitskopien der Beispielprogramme auf der beiliegenden Diskette anlegen. Durch Angabe der jeweiligen Programmnamen laden Sie die Quelltexte in den Arbeitsspeicher, wechseln die Diskette und speichern anschließend die Texte mit END auf der neuen Diskette.

Aufgaben

Jetzt sollten Sie ein wenig in den Kapiteln 4.1 bis 4.3 in der Dokumentation lesen. Dann werden Sie auch wissen, wie Sie aus dem Editor zurückkehren können, ohne daß der Quelltext auf Diskette gespeichert wird, was die Eingabe eines Fragezeichens im Pascal-Menü bewirkt und wie Sie das Listing von PROGRAMM1 auf den Drucker ausgeben können.

Außerdem sollten Sie versuchen, einige der Beispielprogramme (xxx.P) auf der Systemdiskette zu übersetzen.

2 Einführung in Pascal

2.1 Symbole und Syntax-Diagramme

Leider liegt am Anfang Ihrer Arbeit mit Pascal eine *Durststrecke* von einigen Kapiteln, die sich mit etwas abstrakteren Grundlagen beschäftigen. Sollten Sie beim ersten Lesen einige Details nicht ganz verstehen, können Sie später, wenn Sie etwas praktische Erfahrung am Rechner gesammelt haben, diese Teile noch einmal bearbeiten.

Pascal ist eine formale Sprache: Programme sind Folgen von Symbolen. Man kann zwar unendlich viele korrekte Symbolfolgen bilden, jedoch ist die Menge der Regeln, die **Syntax** der Sprache Pascal, endlich.

In diesem Kapitel werden die kleinsten Einheiten von Programmen, die **Symbole** von Pascal, vorgestellt. Diese Symbole sind nicht einzelne Zeichen, sondern

- Bezeichner
- Zahlen
- String-Konstanten
- Sonderzeichen
- Wortsymbole
- Kommentare

Die Definition einer Sprache über Symbole erlaubt eine gewisse Unabhängigkeit von den Eigenschaften spezieller Rechner. So wird z.B. in der Sprache nie Bezug auf Zeilennummern oder die Formatierung des Quelltextes genommen.

Bezeichner bestehen aus einem Buchstaben, gefolgt von Buchstaben oder Ziffern. Ein Bezeichner kann also theoretisch beliebig lang sein.

`VARIABLE B747 ERGEBNIS A B JB007` (alle zulässig)

`3MAL` (das erste Zeichen ist kein Buchstabe)

`ERGEBNIS-3` (Bindestrich nicht erlaubt)

In Pascal 1.4 sind nur die ersten 14 Zeichen signifikant. Somit betrachtet der Compiler die folgenden Bezeichner als gleich:

`EXTRALANGERNAME1 EXTRALANGERNAME2`

Zahlen sind Symbole, die aus komplizierteren Zeichenfolgen bestehen können. Deshalb sind die Bildungsregeln auch nur schwerfällig in Worten zu beschreiben. Eine anschauliche und übersichtliche Beschreibung der Syntax von Pascal liefern die **Syntax-Diagramme**. Bild 2 zeigt die Syntax-Diagramme für Zahlen in Pascal:

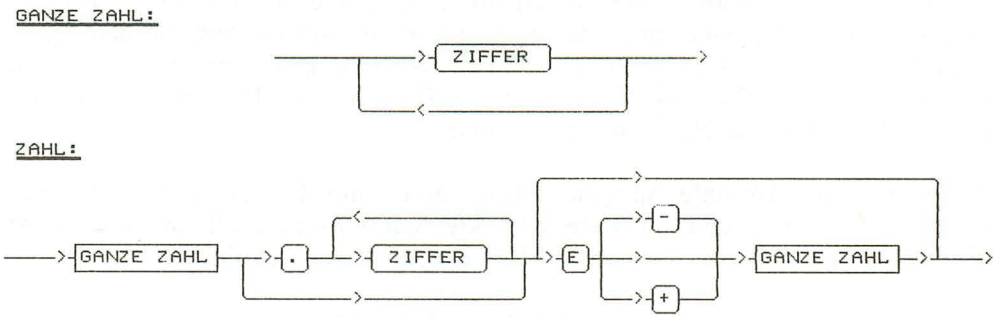


Bild 2: *Zwei Syntax-Diagramme*

Indem man den Pfeilen durch den Graphen folgt und die Zeichen in den Kästen mit den abgerundeten Ecken notiert, erhält man gültige Zahlen in Pascal. Eine *ganze Zahl* besteht also aus einer oder mehreren Ziffern. Im zweiten Diagramm tritt zweimal ein Kästchen mit dem Namen *ganze Zahl* auf. Da die Ecken der Kästchen nicht abgerundet sind, bedeutet dies, daß an diesen Stellen eine Zeichenfolge steht, die durch das Syntax-Diagramm *ganze Zahl* beschrieben wird.

In späteren Kapiteln sollen Sie diese Diagramme selbständig *lesen* können. Alle Syntax-Diagramme für Pascal sind im Anhang A aufgeführt. Für Zahlen wird die Syntax jedoch noch einmal verbal beschrieben, um das Prinzip, das hinter den Diagrammen steht, zu verdeutlichen:

Eine *ganze Zahl* ist eine vorzeichenlose Ziffernfolge. Eine *Zahl* besteht aus einer ganzen Zahl. Daran kann sich ein Dezimalpunkt mit mindestens einer Nachkommastelle anschließen. Danach folgt eventuell der Buchstabe E mit einem Exponent. Der Exponent besteht aus einer ganzen Zahl, der eventuell das Vorzeichen "+" oder "-" vorausgeht.

1 0 1985 0.1 22.3 1E-4 1.5E8	(alle zulässig)
1.	(es muß eine Nachkommastelle folgen)
.1	(es muß eine Null vor dem Punkt stehen)
3,4	(das Komma ist nicht erlaubt)

In Pascal unterscheidet man also zwei **Typen** von Zahlen: Es gibt reelle und ganze Zahlen. Reelle Zahlen sind dadurch gekennzeichnet, daß sie Nachkommastellen und/oder einen Skalierungsfaktor (Exponent) besitzen. Der Skalierungsfaktor gibt an, um wie viele Stellen der Dezimalpunkt verschoben wird:

```
1 = 1E0 = 10E-1 = 100E-2 = 0.1E1 = 0.01E2
```

Eine **String-Konstante** besteht aus einer nicht-leeren Folge von Zeichen, die in Anführungszeichen eingeschlossen ist.

"+-+-" "ABCDE" "Leerzeichen: " (korrekt)

In Standard-Pascal werden Apostrophe und nicht Anführungszeichen verwendet. Dort ist es auch erlaubt, Anführungszeichen in der Zeichenfolge zu verwenden. Pascal 1.4 weicht von dieser Vorgabe ab, da das Betriebssystem des C 64 (z.B. bei der Druckeransteuerung) Anführungszeichen gesondert behandelt.

'alpha'	(Anführungszeichen fehlen)
""	(Strings der Länge Null sind unzulässig)
"klappt's?"	(korrekt)
" !"#$%&"	(Anführungszeichen nicht erlaubt)

In Pascal werden folgende **Sonderzeichen** verwendet:

+	Addition, Vereinigung von Mengen
-	Subtraktion, Differenz von Mengen
*	Multiplikation, Schnitt von Mengen
/	Division
:=	Zuweisung
=	gleich
<>	ungleich
>=	größer oder gleich
<=	kleiner oder gleich
( )	Klammern
[ ]	Index- und Mengenklammern
(* *)	Kommentarklammern
↑	Dereferenzier-Operator
..	Auslassungspunkte
, . ; :	Satzzeichen

Einige Symbole in der Liste bestehen aus zwei Sonderzeichen. Zwischen den beiden Sonderzeichen darf kein Leerzeichen stehen:

:=	(dies ist ein Symbol)
: =	(dies sind zwei Symbole)

Die reservierten **Wortsymbole** von Pascal sind in der folgenden Liste aufgeführt. Sie dürfen nicht als Bezeichner verwendet werden. Ihre Bedeutung wird in den weiteren Kapiteln erklärt:

AND	FILE	NOT	TO
ARRAY	FOR	OF	TYPE
BEGIN	FORWARD	OR	UNTIL
CASE	FUNCTION	PACKED	VAR
CONST	GOTO	PROCEDURE	WHILE
DIV	IF	PROGRAM	WITH
DO	IN	RECORD	
DOWNTO	LABEL	REPEAT	
ELSE	MOD	SET	
END	NIL	THEN	

Im Gegensatz zu BASIC dürfen Bezeichner Wortsymbole enthalten:

```
FORMEL   EINGABEENDE
```

sind also gültige Bezeichner, obwohl sie die Zeichenfolgen FOR, OR und END enthalten.

Da die Größe eines übersetzten Programmes (Objektprogramm) nicht von der Formatierung des Quelltextes abhängig ist, spart man nicht wie in BASIC mit Leerzeichen zwischen den Symbolen. Vielmehr versucht man durch das Layout (Einrückung, Kommentare) die Struktur des Programmes zu unterstreichen. Leerzeichen sind jedoch nur dann syntaktisch erforderlich, wenn durch ihr Fehlen aus zwei Symbolen eines würde.

```
IF X = 6 * Y THEN
```

Hier sind nur zwischen IF und X, sowie zwischen Y und THEN Leerzeichen notwendig.

Kommentare können an jeder Stelle des Programmes eingefügt werden, an der auch ein Leerzeichen stehen darf. Kommentare können beliebige Texte enthalten. Da auf dem C 64 keine geschweiften Klammern vorhanden sind, werden diese durch (* und *) ersetzt. Natürlich darf ein Kommentar nicht die schließende Klammer '*)' enthalten. Andererseits kann sich ein Kommentar über mehrere Zeilen des Quelltextes erstrecken.

Viele Compiler kennen auch *aktive Kommentare*, die den Compilationsvorgang beeinflussen können. In Pascal 1.4 beginnt ein aktiver Kommentar mit einem Dollarzeichen. Die Wirkung aller aktiven Kommentare ist in der Dokumentation beschrieben.

```
(*$R+ Bereichstest einschalten *)
```

2.2 Programmstruktur

In Pascal genügt es nicht, die Befehle des Programmes einfach hintereinanderzustellen. Vielmehr ist - bildlich gesprochen - ein Rahmen erforderlich, der die eigentlichen Anweisungen umgibt. Die Grobstruktur jedes Programmes läßt sich schematisch so angeben:

```
PROGRAM NAME (INPUT,OUTPUT);
  (* Hier ist der Vereinbarungsteil *)
BEGIN
  (* Hier ist der Anweisungsteil    *)
END.
```

Listing 2: *Grobstruktur*

Die erste Zeile ist der **Programmkopf**. Hier wird nach dem Wortsymbol PROGRAM dem Programm ein Name gegeben, der jedoch im Programm keine weitere Bedeutung besitzt. Die Bezeichner INPUT und OUTPUT deuten an, daß das Programm zwei *Kanäle* zur Umwelt besitzt: die Tastatur als Standardeingabe (INPUT) und den Bildschirm als Standardausgabe (OUTPUT). Im Abschnitt 2.16 über Files wird näher auf diese Programmparameter eingegangen.

Bereits im ersten Programm (Listing 1) wurde der Vereinbarungsteil benutzt. Grundsätzlich müssen in Pascal alle Bezeichner definiert werden, bevor sie (in Anweisungen) verwendet werden können.

Im Anweisungsteil eines Programmes stehen die Befehle, die den Algorithmus beschreiben, nach dem die Objekte aus dem Vereinbarungsteil bearbeitet werden.

Bitte achten Sie insbesondere auf die Satzzeichen. Sie sind genauso wichtig wie alle anderen Symbole und dürfen nicht fehlen. Natürlich existiert auch ein Syntax-Diagramm, das den Aufbau eines Programmes definiert. Es heißt PROGRAMM und steht am Ende von Anhang A.

Wenn Sie das Programm in Listing 2 mit diesem Diagramm vergleichen, werden Sie den Pfeilen folgend bis zum Kasten BLOCK gelangen. Dieser bezieht sich auf das Syntax-Diagramm BLOCK. Jeder Weg im Diagramm BLOCK führt vom Eingang links oben zum Ausgang rechts unten über die Symbole BEGIN und END. Auch wenn Sie die vielen Namen in den Kästen noch nicht kennen, ist Ihnen sicherlich klargeworden, daß durch Listing 2 und das Syntax-Diagramm PROGRAMM dieselben Regeln für den **Rahmen** eines Programmes in Pascal definiert werden.

Aufgaben

1. Das Programm Struktur ist ein vollständiges Programm! Lassen Sie es deshalb einmal übersetzen. Experimentieren Sie ein bißchen: Prüfen Sie am Programmbezeichner (NAME) die Regeln für Bezeichner (z.B. STRUKTUR1, 2.PROGRAMM, PROGRAM etc.), entfernen Sie ein paar Satzzeichen (nicht zu viele!) etc. Welche Fehlermeldungen liefert der Compiler? (Erläuterung der Fehlernummern in Anhang B).

2. An welcher Stelle im Syntax-Diagramm PROGRAMM kommt es zu Problemen, wenn Sie folgendes Programm untersuchen?

```
PROGRAM FALSCH (); BEGIN END.
```

   Beheben Sie den Fehler!

2.3 Deklaration von Variablen

Eine Variable läßt sich unter zwei verschiedenen Gesichtspunkten betrachten. Einerseits dient sie zur Programmlaufzeit zur Speicherung von veränderlichen (*variablen*) Werten, wie Zwischenergebnisse oder Zustände des Programmes. Andererseits besitzt sie konstante Eigenschaften: Eine Variable hat einen Namen (Variablenbezeichner), über den sie im Programmtext angesprochen wird. Außerdem kann sie nur eine gewisse Klasse von Werten annehmen (z.B nur Zeichen oder nur Zahlen).

Diese konstanten Eigenschaften werden im Vereinbarungsteil des Programmes für jede im Anweisungsteil benutzte Variable festgelegt. Gewöhnlich wird man Variablen einen Namen geben, der ihre Bedeutung im Programm widerspiegelt:

```
VAR I          : INTEGER;
    ZAEHLER    : INTEGER;
    GEHALT     : REAL;
    DELTA      : REAL;
    ALTER      : INTEGER;
    BUCHSTABE1: CHAR;
    BEFEHL     : CHAR;
```

Listing 3: *Eine Variablendeklaration*

Eine Variablendeklaration beginnt mit dem Wortsymbol VAR. Anschließend werden die Variablenbezeichner aufgeführt. Für jede Variable wird nach einem Doppelpunkt ihr **Typ** angegeben. Dies ist die oben erwähnte Klasse von Werten, die die Variable annehmen kann. In Listing 3 werden die Typen durch Bezeichner (INTEGER, REAL und CHAR) angegeben. An dieser Stelle sei nur soviel gesagt, daß die Variablen I, ZAEHLER und ALTER nur ganze Zahlen, GEHALT und DELTA reelle Zahlen und BUCHSTABE1 und BEFEHL nur Zeichen speichern können.

Die obige Variablendeklaration kann wie folgt abgekürzt werden:

```
VAR I, ZAEHLER, ALTER  :INTEGER;
    GEHALT, DELTA      :REAL;
    BUCHSTABE1, BEFEHL :CHAR;
```

Der entscheidende Vorteil einer expliziten Deklaration jeder Variablen am Programmanfang ist die Möglichkeit, schon während der Übersetzung die Korrektheit von Operationen zu prüfen. Die folgende Zuweisung, die einer Variablen für ganze Zahlen ein Zeichen zuordnen würde, kann sofort vom Compiler als fehlerhaft erkannt werden:

```
ZAEHLER:= BEFEHL
```

Merke: Jeder Bezeichner muß in Pascal vor seiner Anwendung deklariert werden.

2.4 Anweisungen und Ausdrücke

Nun haben Sie endlich das Rüstzeug beisammen, um sich dem Anweisungsteil des Programmes zuzuwenden. Der Anweisungsteil besteht aus einer (wie wir gesehen hatten evtl. sogar leeren) Folge von **Anweisungen** zwischen den Wortsymbolen BEGIN und END.

```
BEGIN
 Anweisung;
 Anweisung;
 ...
 Anweisung;
 Anweisung
END.
```

Die Anweisungen sind voneinander durch Semikola getrennt. In diesem Abschnitt wollen wir die elementarste Form der Anweisung, die **Zuweisung**

vorstellen: Einer Variablen links vom Zuweisungsoperator := wird das Ergebnis der Berechnung des Ausdruckes auf der rechten Seite zugewiesen.

```
I:= 0
I:= I+1
R:= 1/I
```

Da eine Zuweisung den *alten* Wert einer Variablen überschreibt, benötigt man zum Vertauschen der Werte der Variablen I und J eine Hilfsvariable:

`H:=I; I:=J; J:=H` (also nicht `I:=J; J:=I`)

Dies ist auch ein Beispiel für eine Anweisungsfolge.

I+1, 1/I, I, und 0 sind **Ausdrücke**. Im allgemeinen enthalten Ausdrücke mehrere Operanden (Variablen, Konstanten) und Operatoren (+, -, OR, =). Die Struktur eines Ausdruckes wird durch das Syntax-Diagramm AUSDRUCK im Anhang A beschrieben. Deshalb werden hier nicht die formalen Bildungsregeln für Ausdrücke genannt, sondern nur die Besonderheiten von Pascal hervorgehoben.

1. Operatoren dürfen nicht direkt aufeinanderfolgen. Man schreibt also A + (-B) und nicht A + -B.

2. Die Multiplikationsoperatoren *, /, DIV, MOD und AND binden stärker als die Additionsoperatoren +, -, OR (Punkt- vor Strichrechnung).

   `A / 3 + Z` bedeutet `(A / 3) + Z`

   Die Additionsoperatoren binden stärker als die Vergleichsoperatoren =, <>, >=, <=, <, >, IN.

3. Es gibt kein Operationssymbol zum Potenzieren. Der Pfeil ↑ hat eine völlig andere Bedeutung.

4. Eine Folge von Operatoren gleicher Priorität wird von links nach rechts ausgewertet.

   `A * B * C` bedeutet `(A * B) * C`
   `A - B - C` bedeutet `(A - B) - C`

5. Im Zweifelsfall sollte man die Priorität mit Klammern unterstreichen:

`(A * B) - (C * D)` statt `A * B - C * D`

6. Wie in BASIC stehen auch Standardfunktionen zur Verfügung, z.B. SIN, COS, SQRT. Im Augenblick können Sie diese Funktionen *naiv* verwenden. Alle arithmetischen Funktionen sind in der Dokumentation in Abschnitt 4.5 aufgeführt.

Die Operatoren werden später noch detailliert besprochen.

2.5 Einfache Ein- und Ausgabe

Zwar können Sie jetzt bereits korrekte Anweisungsfolgen bilden, jedoch fehlt Ihnen noch eine Anweisung, um die Ergebnisse der Zuweisungen am Bildschirm zu verfolgen. In diesem Abschnitt werden deshalb die Gegenstücke zu PRINT und INPUT in BASIC vorgestellt.

2.5.1 WRITE

In Ihrem ersten Programm (Listing 1) trat bereits die Anweisung WRITELN auf. Syntaktisch gesehen ist WRITELN ein Bezeichner, dem in Klammern **Parameter** folgen können. WRITE und WRITELN bewirken eine Ausgabe auf den Bildschirm.

`WRITE` (Ausdruck)

Dies ist die Grundform einer Ausgabeanweisung. Durch verschiedene zusätzliche Parameter lassen sich die Formatierung und das Ausgabegerät wählen. Wir wollen uns in diesem Abschnitt nur mit der Bildschirmausgabe beschäftigen.

In der oben angegebenen Form hängt die Ausgabe von dem Typ des Ausdruckes ab:

1. Der Ausdruck ist ein String: z.B.

   ```
   WRITE("ADAM & EVA")
   ```

   Der angegebene String wird ab der momentanen Cursorposition ausgegeben. Der Cursor steht danach direkt hinter dem String.

2. Der Parameter ist ein arithmetischer Ausdruck, der also eine Zahl als Ergebnis liefert:

```
WRITE(400+44*2)
```

   Dann wird ab der momentanen Cursorposition der Wert des Ausdruckes (hier also 488) gedruckt. Der Cursor steht nach der Ausführung des Befehls direkt hinter der letzten Ziffer.

3. Schließlich kann der Ausdruck auch ein einzelnes Zeichen als *Ergebnis* besitzen:

```
WRITE("A")
```

   Die Ausgabe erfolgt dann genauso wie bei (1) für einen String der Länge 1. Warum die Fälle (1) und (3) separat aufgeführt werden, wird später in Abschnitt 2.6.2 und 2.9.2 deutlich.

Eine einfache Formatierung der Ausgabe erreicht man, indem man die obigen Parameter um eine Feldgröße nach einem Doppelpunkt erweitert:

```
WRITE("GANZ RECHTS" : 40);
WRITE(30*40 : 10);
```

Die Feldgröße kann ein beliebiger Ausdruck sein, der eine ganze Zahl als Ergebnis liefert. Die Feldgröße bestimmt eine Mindestanzahl an Zeichen, die bei der Ausgabe gedruckt wird.

Ist die Stringkonstante kürzer als die angegebene Feldgröße, so wird der String rechtsbündig in ein Feld der geforderten Länge gestellt. Gleiches geschieht mit einer Zahl, deren Darstellung kürzer als die Feldgröße ist. Die Feldgröße wird ignoriert, falls die Ausgabe zu lang ist (der Punkt steht in den Beispielen für ein Leerzeichen):

```
WRITE(3*4 : 5)  druckt  ...12      (rechtsbündig)
WRITE("***":5)  druckt  ..***      (rechtsbündig)
WRITE(-1E6: 5)  druckt  -1000000   (zu lang)
WRITE("XXX":1)  druckt  XXX        (zu lang)
```

Zwei aufeinanderfolgende WRITE-Befehle können immer zu einem zusammengefaßt werden, wobei man die Parameter durch Kommata trennt. Ein WRITE-Befehl kann also beliebig viele Parameter besitzen:

```
WRITE("DAS FELD IST", A*B: 5, " QUADRATMETER GROSS.");
WRITE(X:5, Y:5, Z:5)
```

Um die nächste Ausgabe in der folgenden Bildschirmzeile fortzusetzen, verwendet man den Befehl WRITELN (write line). Der Befehl WRITELN ohne weitere Parameter setzt den Cursor auf den Anfang der nächsten Bildschirmzeile. Mit der folgenden Befehlsfolge druckt man also drei Leerzeilen.

```
WRITELN; WRITELN; WRITELN
```

Ersetzt man bei (1) bis (3) den Befehl WRITE durch WRITELN, so erfolgt die Ausgabe wie oben beschrieben. Zusätzlich wird am Ende der Ausgabe ein Zeilenvorschub durchgeführt:

```
WRITE  ("DAS FELD IST", A*B: 5);
WRITELN(" QUADRATMETER GROSS.");
WRITELN("↑": 13);
WRITELN("DIES IST ZEILE 3")
```

2.5.2 READ

Haben Sie im Vereinbarungsteil wie im letzten Kapitel beschrieben Variablen deklariert, so können Sie auch vom Bildschirm Werte einlesen. Dabei erfolgt die Eingabe zeilenweise:

Der Benutzer wird mit blinkendem Cursor zu einer Eingabe aufgefordert. Er kann dann beliebige Zeichen eingeben. Schließt er die Eingabe mit der RETURN-Taste ab, so wird die gesamte Bildschirmzeile gespeichert. Der Inhalt der Bildschirmzeile wird mit

```
READ  (Variablenbezeichner)
```

gelesen.

Dabei gibt es folgende Möglichkeiten (Variablendeklaration s. Listing 3):

1. Die Variable ist vom Typ CHAR. Dann wird ein einzelnes Zeichen ab der momentanen Position in der Eingabezeile gelesen und der Variablen zugewiesen.

   ```
   READ (BUCHSTABE1)
   ```

2. Die Variable ist vom Typ INTEGER oder REAL. Zunächst werden vorlaufende Leerstellen überlesen. Nachfolgende Ziffern werden bis zum nächsten Leerzeichen gelesen. Anschließend wird der Variablen der Wert der Ziffernfolge zugewiesen.

```
READ (GEHALT)
```

Wird bei (1) oder (2) das Ende der Eingabezeile erreicht, so wird der Benutzer erneut zur Eingabe einer weiteren Zeile aufgefordert. Wie bei WRITE können zwei aufeinanderfolgende Read-Anweisungen zu einer zusammengefaßt werden. Die zwei folgenden Zeilen liefern also die gleiche Eingabe.

```
READ(BEFEHL); READ(I)
READ(BEFEHL, I)
```

Um bei diesen Eingaben der Variablen BEFEHL das Zeichen "*" und der Variablen I die ganze Zahl 80 zuzuweisen, sind z.B. die folgenden Eingaben möglich:

```
*80          (RETURN-Taste)   oder
*      80    (RETURN-Taste)
```

oder auch in zwei Zeilen:

```
*            (RETURN-Taste)
80           (RETURN-Taste)
```

Gibt man zu viele Werte ein, z.B.

```
*80 90
```

so lesen die obigen Read-Anweisungen bis zum Leerzeichen hinter der Zahl 80. Eine folgende Read-Anweisung wird dann die nächste Zahl - also 90 - lesen. Möchte man den Rest einer Eingabezeile ignorieren, so verwendet man den Befehl READLN (read line). Ohne weitere Parameter überliest er alle Zeichen bis zum Zeilenende. Durch die Anweisungsfolge

```
READ(BEFEHL, I); READLN
```

würde also die Zahl 90 nach der Eingabe von '*' und 80 überlesen werden. Die nächste Read-Anweisung wird also in der nächsten Bildschirmzeile erfolgen. Wie bei WRITELN läßt sich eine solche Folge von READ und READLN zu einem Befehl zusammenfassen:

```
READLN(BEFEHL, I)
```

Abschließend ist noch ein vollständiges Programm mit Ein- und Ausgabe angegeben, das Nullstellen der gemischtquadratischen Funktion x*x+p*x+q bestimmt:

```
PROGRAM PQFORMEL (INPUT, OUTPUT);
  VAR P,Q,W,A: REAL;
BEGIN
  WRITE("P ="); READLN(P);
  WRITE("Q ="); READLN(Q);
  W:= SQRT(P*P/4-Q);
  A:= -P/2;
  WRITELN("X1=",A+W);
  WRITELN("X2=",A-W)
END.
```

Listing 4: *Nullstellenbestimmung*

Aufgaben

1. Damit Sie sich einen praktischen Überblick über die vielen verschiedenen Möglichkeiten zur Ein- und Ausgabe verschaffen können, sollten Sie die Beispiele im Text am C 64 ausprobieren. Dabei dürfen Sie die Deklaration der Variablen und den Rahmen aus Listing 2 nicht vergessen.

2. Ändern Sie das Programm in Listing 4 so, daß die Werte für P und Q vom Benutzer in einer Zeile eingegeben werden und die Ausgabe folgendermaßen formatiert wird:

   ```
   X1=xxxxxxxxxxxx; X2=xxxxxxxxxxxx
   ```

3. Bestimmen Sie die Feldgröße, durch die der Text *Überschrift* auf dem Bildschirm zentriert erscheint. Finden Sie eine allgemeine Formel zur Berechnung der Feldgröße für gegebene Bildschirmbreite und Textlänge!

4. Es werden folgende drei Zeilen eingegeben:

   ```
   1  2  3  4
   5  6  7  8
   9 10 11 12
   ```

   Programmieren Sie drei Anweisungsfolgen, die folgende Werte lesen:

1. Die Zahlen 1, 2, 3, 4 und 9
2. Die Zahlen 1, 5 und 9
3. Die Zahlen 4 und 8

5. Ist Ihnen die genaue Wirkung der Feldgröße noch unklar, sollten Sie folgende Anweisungen programmieren:

```
READ(X, LEN);
WRITELN("!",X:LEN,"!")
```

   Wählen Sie positive und negative Zahlen für X und unterschiedliche Feldgrößen LEN!

6. Experimentieren Sie mit Ausdrücken, Anweisungsfolgen und den Ein- und Ausgabebefehlen! Schreiben Sie kleine Programme, um ein *Gefühl* für Ausdrücke in Pascal zu bekommen!

   Sollten Sie keine eigenen Ideen haben, können Sie sich an der folgenden kleinen Liste orientieren: Berechnung von Zinsen und Zinseszinsen, Berechnung der Fläche von Kreisen und Ellipsen, des Volumens von Kugeln und Kegeln. Berechnung des Logarithmus zur Basis 10. Vergleichen Sie SIN(X)/COS(X) mit TAN(X). Ist SIN(x) * SIN(x) + COS(x) * COS(X)=1 ? Wie berechnet man die Umkehrfunktion von SIN(X)?

2.6 Elementare Datentypen

In den vorausgehenden Abschnitten trat bereits mehrfach der Begriff Typ auf: Zahlen wurden unterschieden in ganze Zahlen und reelle Zahlen, Variablen wurden bei der Deklaration an einen Typ gebunden, und Ausdrücke besaßen einen Typ. Falls Sie bereits in BASIC programmiert haben, wissen Sie, daß es dort nur zwei Typen gibt: (reelle) Zahlen und Strings. In Pascal gibt es auch Objekte von völlig anderen Typen. Später (in Abschnitt 2.12) wird sogar beschrieben, wie man eigene Typen in Pascal definiert.

Dieser Abschnitt beschäftigt sich mit den elementaren Standardtypen INTEGER, REAL, CHAR und BOOLEAN und stellt die Operationen mit Objekten dieser Typen vor.

2.6.1 Der Typ INTEGER

Werte vom Typ INTEGER sind ganze Zahlen, also positive und negative Zahlen ohne Nachkommastellen. Die wichtigsten Operationen, die auf ganze Zahlen anwendbar sind, liefern als Ergebnis wieder eine ganze Zahl:

```
+     Addition
-     Subtraktion oder Vorzeichenwechsel
*     Multiplikation
DIV   ganzzahlige Division
MOD   Modulo-Bildung (Divisionsrest)
```

Um die Wirkungsweise der letzten beiden Operationen zu verdeutlichen, folgen noch einige Zahlenbeispiele:

```
  10  DIV   3  =  3    10  MOD   3  =  1
(-10) DIV   3  = -3  (-10) MOD   3  = -1
  10  DIV (-3) = -3    10  MOD (-3) =  1
(-10) DIV (-3) =  3  (-10) MOD (-3) = -1
```

Formal hängen DIV und MOD wie folgt zusammen:

```
X = (X DIV Y) * Y + (X MOD Y)
```

Weiterhin gibt es die arithmetische Funktion ABS(n), die den Absolutwert (Betrag) der Zahl n liefert.

Jeder Rechner kann nur Zahlen einer endlichen Größe darstellen. In Pascal 1.4 sind als ganze Zahlen nur Werte mit

```
-MAXINT-1 <= n <= MAXINT
```

darstellbar. MAXINT ist ein vordefinierter Konstantenbezeichner, den Sie auch in Ihren Programmen verwenden können. Die Konstante hat den Wert MAXINT = 32787. Tritt bei einer der obigen Operationen mit ganzen Zahlen eine Bereichsüberschreitung auf, so meldet dies das Pascal-Laufzeitsystem und unterbricht das laufende Programm.

2.6.2 Der Typ REAL

Werte des Typs REAL sind reelle Zahlen. Die arithmetischen Operationen (+, -, *, /) liefern angewandt auf reelle Zahlen ein Ergebnis vom Typ REAL. Der Schrägstrich / liefert also das *normale* Ergebnis einer Division:

```
1.5 / 1.2 = 1.25
```

Andererseits dürfen die Operationen MOD und DIV nicht auf Werte vom Typ REAL angewendet werden. Weiterhin sind alle üblichen arithmetischen Funktionen in Pascal definiert. Sie liefern jeweils ein Ergebnis vom Typ REAL:

Bezeichner in Pascal	Bedeutung	Name in BASIC
ABS	Absolutwert	ABS
SQRT	Quadratwurzel	SQR
EXP	Exponentialfkt.	EXP
LN	Nat. Logarithmus	LOG
SIN	Sinus	SIN
COS	Cosinus	COS
ARCTAN	Hauptwert arctan	ATN
SQR	Quadrat	---

Die trigonometrischen Funktionen sind für Winkel in Bogenmaß definiert. Wollen Sie mit Winkeln in Grad arbeiten, so müssen Sie zunächst eine Umrechnung vornehmen. Diese wird als Beispiel für ein Programm mit REAL-Variablen vorgestellt:

```
PROGRAM WINKEL (INPUT, OUTPUT);

  VAR X, T    : REAL;
      FAKTOR1: REAL;

BEGIN
  WRITE("WINKEL:"); READLN(X);
  FAKTOR1:= 1.74532925E-2; (* PI/180 *)
  WRITELN("SIN(",X,")=",SIN(X*FAKTOR1));
  WRITELN;
  WRITE("TANGENS:"); READLN(T);
  WRITE("DER WINKEL",ARCTAN(T)/FAKTOR1);
  WRITELN(" BESITZT DEN TANGENS",T);
END.
```

Listing 5: *Real-Variablen*

In Pascal 1.4 ist die Funktion SQR nicht definiert. Statt dessen sind zwei weitere Funktionen vorhanden, die nicht im Standard vorgesehen sind. Beide Funktionen liefern ein Ergebnis vom Typ REAL:

`POWER(x,y)`	berechnet x hoch y.
`TAN(x)`	berechnet den Tangens von x.

Jede Implementierung setzt auch eine Grenze für den Zahlenbereich, in dem reelle Zahlen dargestellt werden können. Um die Ergebnisse von Operationen mit reellen Zahlen zu verstehen, muß man die interne Darstellung von Werten des Typs REAL kennen.

Wie speichert man z.B. die folgenden Zahlen am günstigsten?

A = 9876543219876543210
B = 1230000000000
C = 0.00000000000234

Die Idee besteht darin, sich zunächst die Größenordnung der Zahl zu merken: A besitzt 19 Stellen vor dem Komma, B hat 13 Vorkommastellen, während C an der 12. Stelle nach dem Komma beginnt. Außerdem werden für jede Zahl möglichst viele Ziffern gespeichert.

Beim C 64 kann eine Zahl maximal 38 Stellen vor oder nach dem Komma beginnen. Von jeder Zahl werden jedoch maximal 9 Ziffern gespeichert. An der 9. Stelle wird gerundet. Intern wird jede Zahl also durch zwei Werte (Mantisse und Exponent) dargestellt:

Mantisse	Exponent
A = 0.987654322	+19
B = 0.123	+13
C = 0.234	-11

Die *Länge* der Mantisse bestimmt also die Genauigkeit, während die Größe des Exponenten die maximale Größe der darstellbaren Zahlen begrenzt.

Merke: Zahlen größer als +/- 10 hoch 38 sind nicht darstellbar.
Zahlen kleiner als +/- 10 hoch -38 werden als 0 dargestellt.
Bei allen Zahlen wird nach der 9. Stelle gerundet.

Diese Grenzen werden in der Praxis mit einem Heimcomputer nie überschritten, außer man wollte z.B. DM-Beträge über 9 Millionen auf den

Pfennig genau speichern. Problematisch ist aber nicht die Speicherung der Zahlen, sondern die Rechnung mit REAL-Zahlen: Da auch alle Zwischenergebnisse *nur* auf neun Stellen genau sind, liefern scheinbar harmlose Berechnungen falsche Ergebnisse:

Mit den obigen Zahlen ist z.B. C + B - B nicht gleich C, da

```
C + B - B = (C + B) - B
= 1.23 E+15 - 1.23 E+15 = 0 <> C
```

Merke: Bei Berechnungen mit REAL-Zahlen immer zuerst Werte der gleichen Größenordnung verknüpfen.

Zwei REAL-Zahlen werden wie folgt auf Gleichheit getestet:

```
IF ABS (A - B) <= EPS THEN ...      und nicht
IF A = B  THEN ...
```

EPS ist dabei ein Wert, der von der Genauigkeit des Rechners abhängt: Beim C 64 wählt man EPS >= 5E-9.

2.6.3 Gegenüberstellung REAL und INTEGER

Solange alle Zwischenergebnisse in dem durch die Konstante MAXINT angegebenen Bereich liegen, sind alle Operationen mit Werten vom Typ INTEGER im mathematischen Sinn *exakt*. Außerdem werden ganze Zahlen kompakter als reelle Zahlen gespeichert: In Pascal 1.4 benötigt eine ganze Zahl nur zwei Speicherstellen gegenüber fünf Speicherstellen für reelle Zahlen. Schließlich sind Operationen auf ganzen Zahlen wesentlich effizienter (kürzerer Code, höhere Ausführungsgeschwindigkeit) als solche mit reellen Zahlen.

Andererseits können große Zahlen nur mit Werten vom Typ REAL dargestellt werden. Auch alle höheren Funktionen liefern Werte vom Typ REAL als Ergebnis.

Zusammenfassend läßt sich sagen, daß reelle Zahlen nur in mathematischen Anwendungen (Nullstellenbestimmungen, Durchschnittswerte etc.) und in kaufmännischen Programmen zur Darstellung großer Zahlen verwendet werden. Typische Anwendungen für ganze Zahlen sind Steuerungsaufgaben im Programm wie Zähler, Indizes und Laufvariablen.

Während Sie schon einige Fälle kennengelernt haben, in denen keine REAL-Werte zulässig sind (Feldgröße bei WRITE oder als Operand bei DIV und MOD), können umgekehrt überall dort, wo reelle Zahlen erlaubt sind, auch ganze Zahlen stehen. Der Compiler erzeugt an diesen Stellen Codes zur Umwandlung in die Darstellung mit Mantisse und Exponent.

Die umgekehrte Umwandlung von reellen Zahlen in ganze Zahlen muß explizit programmiert werden, damit festgelegt werden kann, wie die Nachkommastellen behandelt werden. Im Pascal-Standard sind dazu die Funktionen ROUND und TRUNC vorhanden. ROUND(x) rundet das reelle Argument, während TRUNC nur die Nachkommastellen abschneidet. In Pascal 1.4 ist statt dieser Funktionen die Funktion INT vorhanden, die das reelle Argument zur nächst kleineren ganzen Zahl abrundet. Beispiele zeigen am besten die unterschiedlichen Ergebnisse:

```
ROUND( 3.2)= 3   TRUNC( 3.2)= 3   INT( 3.2)= 3
ROUND( 3.7)= 4   TRUNC( 3.7)= 3   INT( 3.7)= 3
ROUND(-3.2)=-3   TRUNC(-3.2)=-3   INT(-3.2)=-4
ROUND(-3.7)=-4   TRUNC(-3.7)=-3   INT(-3.7)=-4
```

2.6.4 Der Typ CHAR

Nur ein geringer Teil aller Programme arbeitet ausschließlich mit Zahlen. Eine Klasse von Objekten, die vor allem bei der Kommunikation des Rechners mit seiner Umwelt eine große Rolle spielt, sind **Zeichen**.

Werte des Typs CHAR (character) sind einzelne Zeichen. Jedes Zeichen besitzt eine Ordnungsnummer (Codenummer). Der Zusammenhang zwischen Zeichen und Ordnungsnummer ist leider vom jeweiligen Rechner abhängig. Speziell auf Commodore-Rechnern gibt es 256 verschiedene Zeichen. Eine Variable vom Typ CHAR kann also genau eines dieser 256 Zeichen enthalten. Nur 160 dieser Zeichen sind auch am Bildschirm darstellbar. Die restlichen Zeichen (Kontrollzeichen) erfüllen spezielle Aufgaben bei einzelnen Geräten. So besitzen die Funktionstasten bei Tastatureingaben ein eigenes Zeichen, mit einigen Zeichen läßt sich der Cursor am Bildschirm bewegen, und wieder andere Zeichen wählen den Schrifttyp am Drucker.

Konstanten vom Typ CHAR sind einzelne Zeichen, die in Apostrophe (Anführungszeichen in Pascal 1.4) eingeschlossen sind:

```
"-"  "*"  "A"  "x"
```

Vergleiche sind die einzigen Operationen, die zwischen Zeichen definiert sind. Das Ergebnis eines Vergleichs zweier Zeichen ist durch ihre Ordnungszahl festgelegt:

```
"A" < "Z"   "0" < "9"   "!" < ")"
```

Eine Liste aller Zeichen und Codes finden Sie im Handbuch zum C 64 (ASCII- und CHR$-Code). Innerhalb eines Pascal-Programmes können Sie die Ordnungsnummer jedes Zeichens mit der Standardfunktion ORD erhalten.

```
ORD ("A") = 65   ORD("Z") = 90
ORD ("0") = 48   ORD("9") = 57
```

Die Umkehrung der Funktion ORD ist die Funktion CHR: Sie liefert zu einem Argument vom Typ INTEGER das Zeichen mit der angegebenen Ordnungsnummer:

```
CHR (65) = "A"   CHR(57) = "9"
```

Diese Umwandlung zwischen Zeichen und Ordnungsnummer ist, wie bereits erwähnt wurde, vom zugrundeliegenden Zeichensatz abhängig. Eine häufige Anwendung ist der selektive Zugriff auf einzelne Ziffern in einer Zeichenfolge. Das folgende Beispiel soll die Quersumme einer zweistelligen Zahl berechnen, die der Benutzer eingibt

```
PROGRAM SUMME (INPUT, OUTPUT);
  VAR CH1, CH2: CHAR;
      N1,N2   : INTEGER

BEGIN
  READLN(CH1,CH2);
  N1:=ORD(CH1)-ORD("0");
  N2:=ORD(CH2)-ORD("0");
  WRITELN("QUERSUMME ",N1+N2)
END.
```

Listing 6: *Zeichenumwandlung*

Obwohl Sie im Editor zu Pascal 1.4 die Möglichkeit besitzen, mit dem Befehl CHANGE #xxx #yyy direkt ASCII-Codes in eine Stringkonstante einzufügen, sollten Sie die Codeumwandlung explizit im Programm durchführen:

```
WRITE  (CHR(147))
WRITELN(CHR(18),"ERSTE",CHR(146),"TEXTZEILE")
```

2.6.5 Der Typ BOOLEAN

Zur Steuerung des Programmablaufes in Abhängigkeit von bestimmten Bedingungen sind Wahrheitswerte erforderlich. Wahrheitswerte werden in Pascal durch TRUE (wahr) und FALSE (falsch) beschrieben. Formal sind TRUE und FALSE Konstanten vom Typ BOOLEAN. Verschiedene Operationen liefern Wahrheitswerte. Wir hatten bereits die Relationen zwischen Zahlen und Zeichen angesprochen:

```
(17 = 0)        FALSE      "X" < "Y"    TRUE
(17 > 0)        TRUE       "!" < "A"    TRUE
(0.5 = 5E-1)    TRUE       "X" = "x"    FALSE
```

Ein weiteres Beispiel ist die Funktion ODD (n), die den Wert TRUE liefert, falls das Argument n vom Typ INTEGER ungerade ist:

```
ODD (3)         TRUE
ODD (16)        FALSE
ODD (0)         FALSE
```

Entscheidend ist nun, daß man mit diesen Relationen und Funktionen (boolesche) Ausdrücke bilden kann. Sind B1 und B2 zwei boolesche Ausdrücke, so kann man mit den logischen Operatoren AND, OR, NOT neue Ausdrücke bilden.

```
B1 AND B2       TRUE, falls B1=TRUE und  B2=TRUE
B1 OR B2        TRUE, falls B1=TRUE oder B2=TRUE
NOT B1          TRUE, falls B1=FALSE
```

Wenn Sie jetzt noch einmal die Syntax-Diagramme im Anhang A betrachten, werden Sie feststellen, daß dort diese logischen Operatoren mit den arithmetischen Operatoren (+, -, * etc.) aufgeführt sind. AND ist ein Multiplikationsoperator, OR wirkt wie ein Additionsoperator und NOT ist wie ein Vorzeichen definiert.

Diese Definition unterscheidet Pascal von vielen anderen Sprachen, da hierdurch AND, OR und NOT stärker binden als die Relationen =, <, >. Beispiele für Ausdrücke vom Typ BOOLEAN sollen die Unterschiede zeigen:

```
TRUE
ODD(X) OR ODD(Y)
X=Y
CH1="A"
(X=Y) OR (A=B)
ODD(X) AND (X>0) OR ODD(Y) AND (Y>0)
NOT ODD(X) OR (X<0)
(CH>="A") AND (CH<="Z") OR (CH1>="0") AND (CH1<="9")
```

Um die exakten Ergebnisse der Beispiele vorherzusagen, müssen Sie die oben angegebenen Regeln sicher noch einmal genauer studieren. Zur Sicherheit formulieren wir die Prioritätsregeln noch als Merksatz:

Merke: Teilausdrücke, die Vergleiche enthalten, müssen in booleschen Ausdrücken geklammert werden. AND bindet stärker als OR.

Interessant werden Variablen vom Typ BOOLEAN erst in großen Programmen. Mit ihnen kann man Zustände im Programmablauf beschreiben. Nach der Variablendeklaration

```
VAR P, Q, SPEICHERLEER:   BOOLEAN;
    ALLESFALSCH, ZUGROSS: BOOLEAN;
    ZAEHLER,I:            INTEGER;
```

kann man folgende Operationen durchführen:

```
P:= TRUE; Q:= P;
SPEICHERLEER:= ZAEHLER<=0; ZUGROSS:= I>=250;
ALLESFALSCH:= SPEICHERLEER AND ZUGROSS;
IF ALLESFALSCH THEN ...
IF SPEICHERLEER AND NOT ALLESFALSCH THEN ...
```

Dieses Beispiel verdeutlicht auch eine Namenskonvention: Man bezeichnet boolesche Variablen meist mit Adjektivnamen. Nur selten wird die Tatsache ausgenutzt, daß die Funktion ORD auch auf boolesche Argumente anwendbar ist. Dadurch ist auch der Typ BOOLEAN geordnet. Es gilt:

```
ORD (FALSE) = 0    ORD (TRUE) = 1
FALSE < TRUE
```

Eine Bitte am Schluß: Schreiben Sie in einem booleschen Ausdruck nicht

```
P = TRUE     oder     SPEICHERLEER = FALSE
```

Dies ist zwar völlig korrekt, zeugt aber von einem schlechten Stil. Man schreibt einfacher und deutlicher:

```
P            oder     NOT SPEICHERLEER
```

Aufgaben

1. Prüfen Sie die Bereichsgrenzen für reelle und ganze Zahlen in Pascal 1.4. Welche Fehlermeldungen erhalten Sie? Lokalisieren Sie die Fehler im Quelltext mit der Option LOCATE ADDRESS!

2. Wie muß man in Abschnitt 2.62 den Ausdruck C+B-B klammern oder umstellen, um ein korrektes Ergebnis zu erhalten?

3. Schreiben Sie einen Ausdruck mit der Funktion INT, der eine reelle Zahl R wie die Funktion ROUND rundet. (Zur Not finden Sie den Ausdruck in der Dokumentation in Kapitel 4)

4. Beweisen Sie durch Einsetzen aller möglichen Kombinationen von TRUE und FALSE, daß die folgenden booleschen Ausdrücke äquivalent sind (Gesetze von de Morgan):

   `NOT(A AND B)` entspricht `NOT(A) OR NOT(B)`

   `NOT(A OR B)` entspricht `NOT(A) AND NOT(B)`

2.7 Deklaration von Konstanten

Oft gibt es gewisse Werte in einem Programm, die während der gesamten Laufzeit des Programmes nicht verändert werden. Für diese **Konstanten** kann man in Pascal Bezeichner vergeben. Wie die Variablendeklaration muß die Konstantendeklaration im Vereinbarungsteil erfolgen. Wie aus dem Syntax-Diagramm BLOCK im Anhang A zu entnehmen ist, steht die Konstantendeklaration vor der Variablendeklaration:

```
PROGRAM KONSTANTEN (OUTPUT);
CONST FAKTOR1      =1.745329252E-2; (* PI/180  *)
      FAKTOR2      =57.29577951;    (*1/FAKTOR1*)
      CLEARSCREEN  =147
      ENDEKOMMANDO ="*"
      VERSION      ="VERSION 747";

BEGIN
  WRITELN(CHR(CLEARSCREEN), "DIES IST ",VERSION);
  WRITELN(FAKTOR1, 1/FAKTOR2)
END.
```

Listing 7: *Konstantendeklaration*

Eine Konstantendeklaration wird durch das Wortsymbol CONST eingeleitet. Jedem Bezeichner wird nach einem Gleichheitszeichen ein Wert zugeordnet.

Der Typ des Wertes bestimmt auch den Typ des Konstantenbezeichners. VERSION ist also eine Stringkonstante, während FAKTOR2 vom Typ REAL ist. Nach dem Gleichheitszeichen darf nur eine Konstante (evtl. mit Vorzeichen) folgen. Andere Ausdrücke sind nicht erlaubt:

```
CONST FAKTOR2 = 1/FAKTOR1;
```
(falsch!)

2.8 Kontrollstrukturen

Bisher haben wir nur lineare Programme vorgestellt, das sind Programme, in denen die Anweisungen genau in der Reihenfolge ausgeführt werden, in der sie im Programmtext stehen. Eine entscheidende Fähigkeit von Rechnern ist jedoch gerade die Möglichkeit, Anweisungen zu wiederholen oder in Abhängigkeit von Bedingungen auszuführen, die während des Programmes geprüft werden. In BASIC wird dies durch Sprünge im Programm erreicht (IF, GOTO, FOR ... NEXT, ON ... GOTO).

In diesem Abschnitt werden die entsprechenden Kontrollstrukturen in Pascal vorgestellt. Bedingungen und Wiederholungen werden dort durch sogenannte zusammengesetzte Anweisungen gebildet, die dem Programm eine **Blockstruktur** geben.

Diese Blockstruktur ist Grundlage für ein fundamentales Prinzip der Strukturierten Programmierung: Der (statische) Programmtext zeigt bereits die dynamische Struktur (z.B. Schleifen) des Programmes. Das Programm besteht aus Blöcken, die jeweils nur einen Eingang und einen Ausgang besitzen. Blöcke dürfen (nach genauen Regeln) zu einem Block zusammengefaßt werden.

In Pascal bezeichnet man einen solchen Block als eine **Anweisung**. In den folgenden Abschnitten werden Sie die obigen prinzipiellen Aussagen über Blöcke in den Syntax-Regeln von Anweisungen in Pascal wiederfinden.

2.8.1 Anweisungsfolgen

Die elementaren Bausteine eines Programmes sind die einfachen Anweisungen. Beispiele für einfache Anweisungen kennen Sie bereits aus den Abschnitten 2.4 und 2.5. Dort wurden die Zuweisung und die Ein- und Ausgabeanweisungen vorgestellt:

```
A:= A+1
WRITE(A)
READLN
```

Dort wurde auch erwähnt, daß der Anweisungsteil aus einer Folge von Anweisungen besteht, die mit BEGIN und END gekennzeichnet wird. Dabei werden die Anweisungen durch Semikola getrennt:

```
BEGIN
  Anweisung;
  Anweisung;
  ...
  Anweisung
END
```

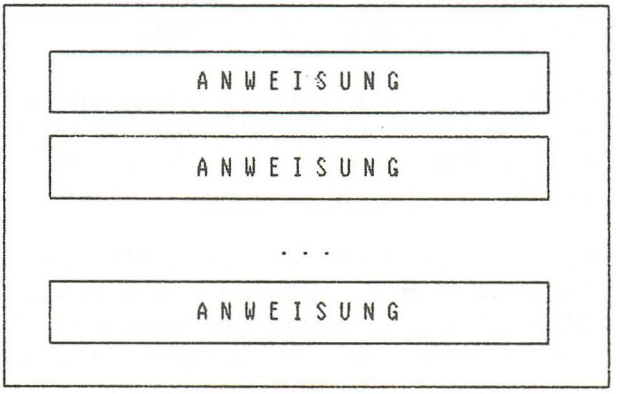


Bild 3: *Anweisungsfolge*

Bild 3 zeigt Ihnen die Struktur einer Anweisungsfolge. Daneben ist zur Verdeutlichung die Blockstruktur skizziert: Die einzelnen Anweisungen betrachtet man als Blöcke, die durch die Wortsymbole BEGIN und END zu einer Anweisung (Block) *geklammert* werden. Diese Klammerung werden wir später benutzen, um in zusammengesetzten Anweisungen, bei denen eine einzelne Anweisung erwartet wird, eine ganze Anweisungsfolge einzusetzen.

Noch ein Wort zu den Semikola: Ein Semikolon **trennt** Anweisungen. Deshalb ist kein Semikolon vor dem abschließenden END erforderlich. Setzen Sie dort auch ein Semikolon,

```
BEGIN WRITE(A); A:= A+1; WRITE(A); END
```

so erwartet der Compiler eine Anweisung. Um diesen Fall nicht als Fehler zu behandeln, gibt es in Pascal die Leeranweisung, die aus keinem Befehl besteht. Diese mysteriöse Anweisung tritt auch in den folgenden (korrekten) Anweisungsfolgen auf:

`BEGIN END`	(1 Leeranweisung)
`BEGIN A:=B; END`	(1 Leeranweisung)
`BEGIN ; ; END`	(3 Leeranweisungen)

2.8.2 Bedingte Anweisungen

Eine bedingte Anweisung (If-Anweisung) hat die folgende Form:

```
IF Ausdruck THEN
  Anweisung 1
ELSE
  Anweisung 2
```

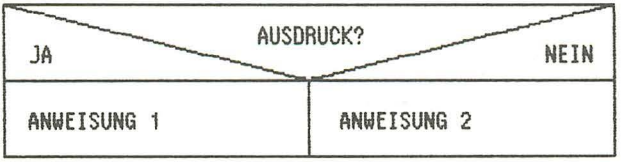


Bild 4: *If-Anweisung*

Der Ausdruck muß ein Ergebnis vom Typ BOOLEAN liefern. Ist das Ergebnis TRUE, wird die Anweisung nach dem Wortsymbol THEN ausgeführt. Ist das Ergebnis FALSE, so wird die Nein-Anweisung ausgeführt. Es gibt viele verschiedene Formen, die If-Anweisung im Quelltext zu formatieren. In diesem Buch wird das folgende Layout verwendet:

```
IF KONTO>=0 THEN
   WRITELN( KONTO:8,"DM GUTHABEN")
ELSE
   WRITELN(-KONTO:8,"DM SCHULDEN")
```

Bei einer anderen Form der If-Anweisung ist keine Nein-Anweisung vorgesehen. Nur wenn der Ausdruck den Wert TRUE liefert, wird die Anweisung nach dem Wortsymbol THEN ausgeführt:

```
IF Ausdruck THEN
  Anweisung
```

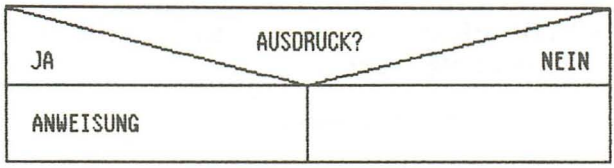


Bild 5: *If-Anweisung*

```
IF A<B THEN
   BEGIN H:=A; A:=B; B:=H END
```

Dieses Beispiel zeigt auch, wie man statt einer einzelnen Anweisung eine ganze Anweisungsfolge durch eine Bedingung kontrolliert. Der Compiler kümmert sich nämlich nicht um die Einrückungen im Quelltext. Deshalb würde er das folgende Programm nicht korrekt übersetzen:

```
IF A>B THEN
   MAX:= A; MIN:= B
ELSE
   MAX:= B; MIN:= A
```

Nach der oben angegebenen Syntax der If-Anweisung muß nach THEN und ELSE eine einzelne Anweisung folgen. Deshalb *sieht* der Compiler den folgenden Text:

```
IF A>B THEN
   MAX:= A;
MIN:= B; ELSE
MAX:= B; MIN:= A
```

Um die Struktur, die durch die Einrückung beschrieben wird, auch korrekt in Pascal zu formulieren, muß man also die Anweisung MAX:=A; MIN:= B zu einer Anweisungsfolge zusammenfassen:

```
IF A>B THEN
    BEGIN MAX:= A; MIN:= B END
ELSE
    BEGIN MAX:= B; MIN:= A END
```

Auf einen weiteren Fallstrick müssen Sie noch achten: Hätten wir vor dem Wortsymbol ELSE (also nach dem END) ein Semikolon gesetzt, würde der Compiler eine If-Anweisung wie in Bild 5 erkennen und damit das ELSE nach dem Semikolon als Fehler markieren.

Am Beispiel der Bedingten Anweisung wollen wir noch das Prinzip der Schachtelung von Anweisungen (Blöcken) zeigen. Eine If-Anweisung ist eine zusammengesetzte Anweisung. Dadurch werden die Anweisungen nach THEN und ELSE zu **einer** Anweisung zusammengefaßt. Dies soll auch der äußere Rahmen in den Abbildungen 4 und 5 unterstreichen. Damit kann also eine If-Anweisung selbst als Teil einer anderen If-Anweisung auftreten. Um das Maximum von drei Zahlen zu berechnen, kann man folgende Anweisung verwenden:

```
IF A>B THEN
  IF A>C THEN
    MAX:=A
  ELSE
    MAX:= C
ELSE
  IF B>C THEN
    MAX:= B
  ELSE
    MAX:= C
```

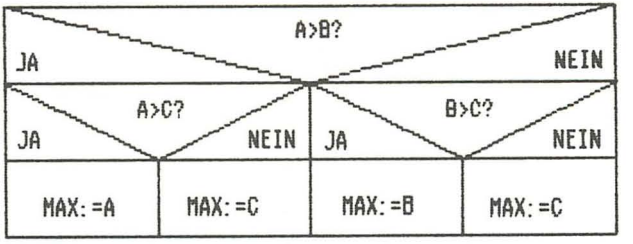


Bild 6: *Geschachtelte Blöcke*

Diese Anweisung ist korrekt, jedoch sollte man solche geschachtelten If-Anweisungen *sicherheitshalber* mit BEGIN END klammern, da sonst evtl. die ELSE-Teile ungewollt falsch gegliedert werden. Der Compiler ordnet nämlich jedes ELSE der letzten If-Anweisung zu, die noch kein ELSE besitzt. Dies führt im folgenden Beispiel zu Problemen.

```
IF ZEILEBEENDET THEN
   IF ZEILENNUMMER=5 THEN WRITE("ZEILE 5")
ELSE
   WRITELN ("KEIN ZEILENENDE")
```

Das ELSE wird hier der Bedingung IF ZEILENNUMMER=5 zugeordnet, was sicher nicht die Absicht des Programmierers war. Um das korrekte Ergebnis zu erhalten, muß die If-Anweisung zwischen THEN und ELSE mit BEGIN und END geklammert werden.

2.8.3 Fallunterscheidung

In manchen Fällen muß man in Abhängigkeit eines einzigen Wertes unterschiedliche Operationen vornehmen. In diesem Fall bietet sich statt einer Folge von If-Anweisungen die Case-Anweisung an:

```
CASE Ausdruck OF
  Fallmarken 1: Anweisung 1;
  Fallmarken 2: Anweisung 2;
  ...
  Fallmarken n: Anweisung n
END
```

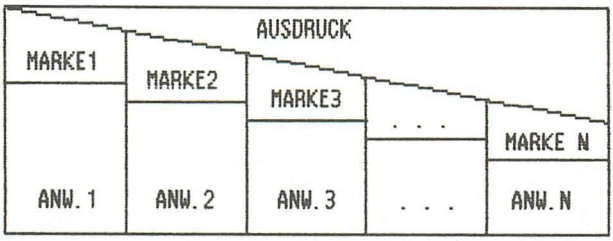


Bild 7: *Fallunterscheidung*

Die exakte Syntax ist dem Syntax-Diagramm FALLUNTERSCHEIDUNG im Anhang A zu entnehmen. Eine Fallunterscheidung wird folgendermaßen ausgeführt: Zunächst wird der Ausdruck ausgewertet. Er muß einen Wert eines skalaren Typs (z.B. CHAR, INTEGER, aber nicht REAL) liefern. Von den nachfolgenden Anweisungen wird nur diejenige ausgeführt, die den Wert des Ausdruckes in ihrer Konstantenliste enthält. Natürlich müssen die Fallmarken denselben Typ wie der Ausdruck besitzen. Kommt der Wert des Ausdruckes in keiner Fallmarke vor, so erfolgt ein Programmabbruch mit Fehlermeldung. Im Zusammenhang mit Aufzählungstypen und Varianten Records wird sich die Case-Anweisung als besonders nützlich erweisen. Das folgende Beispiel zeigt eine typische Anwendung der Case-Anweisung:

```
PROGRAM FALL (INPUT, OUTPUT);
VAR CH          : CHAR;
    A, B, ERG   : REAL;
    OK          : BOOLEAN;

BEGIN
  READLN(CH, A, B);
  OK:= TRUE;
  CASE CH OF
   ".","*" : BEGIN ERG:= A * B END;
   "/":      BEGIN
              IF B = 0 THEN
                BEGIN
                 OK:= FALSE;
                 WRITELN("DIVISION DURCH NULL")
                END
              ELSE
                ERG:= A / B
             END;
   "+"     : ERG:= A + B;
   "-"     : ERG:= A - B
   ELSE      BEGIN OK:= FALSE;
               WRITELN("'",CH,"' NICHT ERLAUBT!")
             END
  END;
  IF OK THEN WRITELN("ERGEBINS",ERG)
END.
```

Listing 8: *Beispielprogramm*

Das Programm benutzt bereits eine Erweiterung der Case-Anweisung in Pascal 1.4: Durch die Angabe des Wortsymbols ELSE am Ende der Case-Anweisung kann eine Anweisung genannt werden, die in dem Fall durchgeführt wird, wenn der Wert des Ausdruckes mit keiner Fallmarke übereinstimmt. Dann wird natürlich auch kein Programmabbruch mit Fehlermeldung durchgeführt.

Bitte beachten Sie die Verwendung der booleschen Variablen OK. Da die Case-Anweisung nur einen *Ausgang* besitzt, muß das Auftreten eines Fehlers im Inneren der Case-Anweisung explizit notiert werden.

Zum Schluß sei darauf hingewiesen, daß manche Compiler (nicht Pascal 1.4) Fallmarken aus einem zusammenhängenden Wertebereich erwarten. Sie würden einen für folgendes Beispiel sehr ungünstigen Code erzeugen:

```
CASE GANZEZAHL OF
 2    : BEGIN END;
 200  : BEGIN END;
 2000 : BEGIN END
END
```

2.8.4 While-Anweisung

Möchte man eine Anweisung (einen Block) wiederholt ausführen, so gibt es dafür in Pascal drei verschiedene zusammengesetzte Anweisungen, die jeweils unterschiedliche Kontrollstrukturen bilden. Die Unterschiede bestehen in der Form, in der die Bedingung formuliert wird, unter der die Anweisung wiederholt wird. Dieser Abschnitt beschreibt die am häufigsten benutzte Wiederholungsanweisung.

Die While-Anweisung besitzt die folgende Struktur:

```
WHILE Ausdruck DO
  Anweisung
```

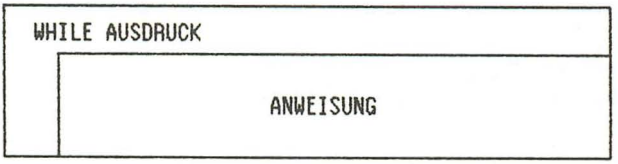


Bild 8: *While-Anweisung*

Der Ausdruck liefert ein Ergebnis vom Typ BOOLEAN. Die Wiederholung wird wie folgt ausgeführt:

1. Der boolesche Ausdruck wird ausgewertet. Ist das Ergebnis FALSE, so wird die gesamte While-Anweisung beendet.

2. Ist das Ergebnis des Ausdruckes TRUE, so wird die Anweisung nach dem Wortsymbol DO ausgeführt. Anschließend wird die Ausführung bei 1. fortgesetzt.

Ein Beispiel soll die Anwendung der Struktur erklären. Wir wollen ohne Logarithmusfunktion bestimmen, wie viele Stellen die Zahl X vor dem Komma besitzt.

```
PROGRAM STELLEN(INPUT, OUTPUT);
  VAR X: INTEGER; N: INTEGER;

BEGIN
  READLN(X); N:=0
  WHILE X<>0 DO
    BEGIN
     X:= X DIV 10; N:= N+1
```

```
    END;
  WRITELN("ANZAHL DER STELLEN:",N)
END.
```

Die Idee besteht also darin, zu zählen, wie oft man die Zahl nach *rechts schieben* kann, bis alle Ziffern hinter dem Komma stehen. Wichtig ist dabei, daß die Prüfung des Ausdruckes (X<>0) **vor** der Ausführung der Anweisung erfolgt. Deshalb wird für die Eingabe von 0 die Schleife überhaupt nicht durchlaufen. Also liefert das Programm für diese Eingabe das Ergebnis N=0. Anzumerken ist noch, daß das Programm auch für negative Zahlen korrekt arbeitet.

Nach der ausführlichen Diskussion im letzten Abschnitt ist Ihnen sicher auch klar, warum im Programm STELLEN nach dem Wortsymbol DO eine Anweisungsfolge (BEGIN ... END) steht. Ist dies nicht der Fall, sollten Sie das Programm probeweise ohne die Klammerung mit BEGIN und END übersetzen und testen. Wenn Sie anschließend die beiden letzten Kapitel noch einmal lesen, werden Sie die Bedeutung der Anweisungsfolge zur Bildung von Blöcken erkennen.

While-Anweisungen nennt man auch *pre check loops* oder abweisende Schleifen. Durch die Eigenschaft einer While-Anweisung, die Schleife auch nicht auszuführen, spart man oft notwendige Sonderbehandlungen. Das folgende Beispielprogramm berechnet für zwei natürliche Zahlen N und K den Wert E = N hoch K.

```
E:= 1; I:= K;
WHILE I>0 DO
  BEGIN E:= E*N; I:= I-1 END
```

Dieses Beispiel berücksichtigt die Sonderfälle N=0 und K=0 korrekt. Es gilt nämlich

N hoch 0 = 1	für alle N
0 hoch K = 0	für alle K<>0

Auch für die Wiederholungsanweisungen, die wir erst im nächsten Abschnitt kennenlernen, gelten die folgenden Regeln, die man bei der Programmierung beachten sollte:

1. Bei der Wiederholung muß die Schleife irgendwann beendet werden. Deshalb muß man sich während der Berechnung dem *Ziel* nähern. Folgendes Programmstück ist also auf jeden Fall sinnlos:

```
WHILE I<>0 DO K:= K+1
```

2. Während jeder Ausführung der Wiederholung müssen gewisse Bedingungen erhalten bleiben. Diese Bedingungen bezeichnet man auch als Schleifen-Invarianten. Es ist keine schlechte Idee, diese Invarianten durch Kommentare zu verdeutlichen:

```
E:= 1; I:= K;
WHILE I>0 DO  (* E = X hoch (K-I) *)
  BEGIN E:= E*N; I:= I-1 END
```

   Am Ende der Schleife ist also I=0, und damit besitzt E den gewünschten Wert.

3. Grundsätzlich soll man beim Entwurf einer Wiederholung den *Sonderfällen* besondere Aufmerksamkeit schenken, um logische Fehler nicht erst im fertigen Programm bei seltenen Eingaben zu finden.

2.8.5 Repeat-Anweisung

Seltener als die While-Anweisung wird die Repeat-Anweisung benutzt. Sie hat die in Abbildung 9 angegebene Struktur.

```
REPEAT
  Anweisung;
  Anweisung;
  ...
  Anweisung
UNTIL Ausdruck
```

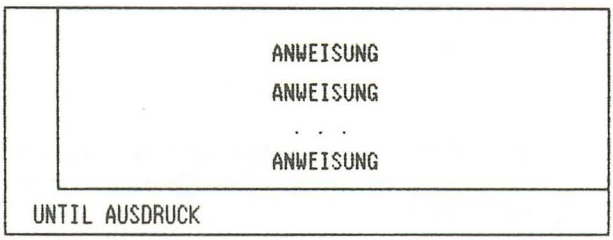


Bild 9: *Repeat-Anweisung*

Der wesentliche Unterschied zur While-Anweisung ist die Tatsache, daß die Anweisungsfolge mindestens einmal ausgeführt wird:

1. Die Anweisungsfolge wird ausführt.

2. Anschließend wird der Ausdruck vom Typ BOOLEAN ausgewertet. Ist das Ergebnis TRUE, so wird die Ausführung der Repeat-Anweisung beendet. Ist das Ergebnis FALSE, so wird die Ausführung bei 1. fortgesetzt.

Im Gegensatz zur While-Anweisung kann der boolesche Ausdruck also Variablen enthalten, die erst innerhalb der Anweisungsfolge berechnet werden.

```
REPEAT
  WRITE ("ALLES KLAR? (J,N) ");
  READLN (CH)
UNTIL (CH="J") OR (CH="N")
```

Weitere Beispiele werden in den folgenden Abschniiten vorgestellt.

2.8.6 For-Anweisung

In BASIC bietet die For-Anweisung die beste Möglichkeit zur Strukturierung von Wiederholungen. In Pascal wird die For-Anweisung nur dann verwendet, wenn die Anzahl der Wiederholungen bereits **vor** Eintritt in die Schleife bekannt ist. Die Struktur der Anweisung ist wieder in einem Bild dargestellt:

```
FOR Variable := Ausdruck TO Ausdruck DO
  Anweisung
```

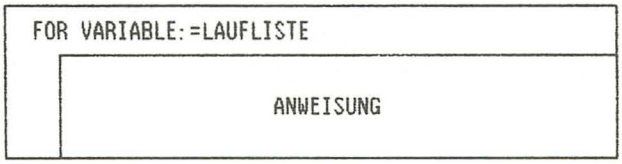


Bild 10: *For-Anweisung*

Die Ausführung erfolgt nach dem folgenden Schema. Es sichert, daß der Endwert der Schleife nicht in der Schleife verändert werden kann. Außerdem kann die For-Schleife wie die While-Schleife auch nicht durchlaufen werden, falls nämlich der Wert von Ausdruck 1 bereits größer als der Wert von Ausdruck 2 ist:

1. Ausdruck 1 wird ausgewertet. Der Wert wird der Laufvariablen zu-gewiesen. Anschließend wird Ausdruck 2 ausgewertet und als Endwert der Schleife gespeichert.

2. Der Wert der Laufvariablen wird mit dem gespeicherten Endwert verglichen. Ist er größer als der Endwert, so wird die Ausführung der Schleife beendet.

3. Sonst wird die Anweisung nach dem Wortsymbol DO ausgeführt.

4. Anschließend wird die Laufvariable um 1 erhöht und die Ausführung der Schleife bei 2. fortgesetzt.

Die folgende Anweisungsfolge berechnet die Summe der Kehrwerte der Zahlen von 1 bis 100.

```
S:= 0.0;
FOR I:= 1 TO 100 DO S:= S+1/I
```

Als Typ der Laufvariablen sind alle skalaren Typen (z.B. INTEGER, CHAR, aber **nicht** REAL) zulässig: Dementsprechend wird in Schritt 4 allgemein der Nachfolger im Wertebereich bestimmt.

```
FOR CH:= " " TO "Z" DO
   WRITELN("DER CODE VON ",CH," IST",ORD(CH):4)
```

Die einzige Möglichkeit, die Schrittweite zu ändern, besteht darin, rückwärts zu zählen: Man ersetzt das Wortsymbol TO durch das Wortsymbol DOWNTO. Das Schema (1.-4.) gilt analog. Um andere Schrittweiten als +1 und -1 zu erhalten, muß man die Wiederholung explizit mit der While-Anweisung programmieren. Besonders vorsichtig muß man dabei bei Laufvariablen vom Typ REAL sein. Addiert man ständig eine Schrittweite, so können sich die Rundungsfehler aufschaukeln. Soll z.B. die reelle Variable W die Werte 1 0.9 0.8 0.7 ... -0.9 -1 durchlaufen, so schreibt man nicht

```
W:= 1.0;
WHILE W>=(-1) DO
  BEGIN
    WRITELN(W); W:= W-0.1
  END
```

sondern

```
FOR I:= 10 DOWNTO -10 DO
  BEGIN
    W:= I/10; WRITELN(W)
  END
```

Die Erstellung einer Multiplikationstabelle ist ein anschauliches Beispiel für geschachtelte For-Schleifen.

```
PROGRAM MULTIPLIKATION(OUTPUT);
  CONST N=13;
  VAR I,J : INTEGER;
BEGIN
  FOR I:= 1 TO N DO
    BEGIN
      FOR J:= 1 TO N DO
        WRITE(I*J:3);
      WRITELN
    END
END.
```

2.8.7 Sprunganweisung

In diesem Abschnitten werden alle Regeln, die die Sprunganweisung betreffen, zusammengestellt. Deshalb werden einige Begriffe auftreten, die erst in späteren Abschnitten über Prozeduren erklärt werden.

Die Sprunganweisung paßt eigentlich nicht in das Blockkonzept der Sprache Pascal. Darum ist bei ihrer Verwendung auch die Einhaltung einiger Nebenbedingungen erforderlich.

Eine Sprunganweisung hat die folgende Form:

```
GOTO Label
```

Die Programmausführung wird beim Erreichen des Labels an der Anweisung fortgesetzt, die durch das Label markiert wird. Labels sind positive ganze Zahlen. Jede Anweisung kann durch ein Label markiert werden:

```
Label: Anweisung
```

Alle Labels müssen außerdem in dem Block, in dem sich die markierte Anweisung befindet, im Labeldeklarationsteil aufgeführt werden. Eine Labeldeklaration ist die erste Deklaration in einem Block und hat die folgende Form:

```
LABEL Label1, Label2, ..., LabelN;
```

Bei Sprüngen muß die Blockstruktur berücksichtigt werden:

1. Es ist nicht erlaubt, von außen in eine Prozedur zu springen.

2. Es ist nicht erlaubt, von außen in eine For-, With- und Case-Anweisung zu springen.

Jedoch ist es erlaubt, aus einer geschachtelten Prozedur zu einer Marke in einem umfassenden Block zu springen.

Sprunganweisungen sollten nur zur Behandlung selten auftretender Ausnahmefälle benutzt werden. Außerdem ist es sinnvoll, im Labeldeklarationsteil Kommentare einzuführen, die die Bedeutung des Labels erklären.

Aufgaben

1. Schreiben Sie Programme zur Umwandlung arabischer Zahlen in römische Zahlen. Vielleicht haben Sie auch eine Idee, wie man die Umwandlung in umgekehrter Richtung vornehmen kann.

2. Schreiben Sie ein Programm, das Zahlenfolgen einliest und bei der Eingabe der Zahl 999 die Summe über die Folge (natürlich ohne 999) druckt. Welche Wiederholungsanweisung ist hier angebracht?

3. Geben Sie eine Anweisungsfolge an, die für beliebige X- und Y-Werte einen Cursor in Zeile X und Spalte Y positioniert.

4. Erstellen Sie ein Programm, das die Nullstellen einer Funktion F nach dem Intervallschachtelungsverfahren bestimmt: Der Benutzer gibt als Startwerte die Intervallgrenzen L und R an, in denen eine Nullstelle liegt. Dabei soll gelten F(L)*F(R)l=0, d.h. zwischen L und R liegt ein Vorzeichenwechsel. Die Nullstelle wird durch sukzessive Intervallhalbierung gefunden. In Abhängigkeit vom Funktionswert in der Intervallmitte M=(L+R)/2 wird M als rechte oder linke Intervallgrenze benutzt. Die Approximation beende man, falls die Intervallgröße kleiner als eine vom Benutzer vorgegebene Genauigkeit ist. Prüfen Sie das Programm mit der Funktion F:=2*SIN(X)-COS(2*x) im Intervall 0 bis 1.

5. Schreiben Sie ein Programm, das alle Primzahlen in einem vom Benutzer vorgegebenen Intervall druckt. (Eine Zahl X heißt Prim, wenn sie nur durch 1 und sich selbst geteilt werden kann. 1 ist keine Primzahl.) Überlegen Sie sich vor der Programmierung, welche Zahlen von Anfang an als Primzahlen ausscheiden und bis zu welcher Grenze die Teilbarkeit geprüft werden muß.

6. Schreiben Sie ein Programm, das eine Wahrheitstabelle in der folgenden Form druckt:

A	B	A AND B
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

   Geben Sie sich ruhig etwas Mühe bei der Formatierung. Benutzen Sie (String-)Konstanten! Besonders schön wäre es, wenn Sie zwei geschachtelte Schleifen mit booleschen Variablen A und B verwenden würden.

   Prüfen Sie mit dem Programm die Ergebnisse von A<B, A>B, A<=B, A>=B statt A AND B!

7. Überlegen Sie, wie man die Operation I MOD 7 verwenden kann, um die Ausgabe von Ergebnissen so zu steuern, daß jeweils sieben Werte in einer Zeile stehen.

8. Schreiben Sie ein Programm, das so lange Zeichen einliest, bis es auf eine Ziffer trifft, und dann **zeichenweise** eine Zahl einliest und den Wert der Zahl in der Variablen R abspeichert.

   `abcdef12.34ghijk` liefert den Wert R=12.34

   Erweitern Sie das Programm so, daß es auch einen Skalierungsfaktor (E+xx) einlesen kann! Vielleicht hilft Ihnen bei der Programmierung das Syntax-Diagramm ZAHL im Anhang A.

2.9 Die Datenstruktur Array

In der Praxis kommt man mit den bisher besprochenen einfachen Datentypen nicht aus. Will man z.B. auf einer Menge gleichartiger Werte die gleiche Operation wiederholen, so müßte man für jedes Objekt einen eigenen Bezeichner definieren und die gleiche Operation mit jedem Wert einzeln durchführen.

Für solche Probleme vereinbart man - anschaulich gesprochen - eine Tabelle aller Werte. Nur die Tabelle erhält einen Bezeichner, so daß jeder einzelne Wert über den Bezeichner der Tabelle zusammen mit einem **Index** in der Tabelle angesprochen wird. Einen solchen Datentyp nennt man in Pascal ein **Array**.

Die einfachste Form eines Arrays besitzt als Elemente einzelne Werte. Dabei spielen Arrays von Zeichen eine besondere Rolle. (Sie entsprechen etwa den Strings in BASIC.) Im Abschnitt 2.9.3 werden wir Arrays betrachten, deren Elemente wiederum Arrays sind.

2.9.1 Eindimensionale Arrays

Eindimensionale Arrays sind Arrays, die als Elemente nicht wieder Arrays besitzen.

```
CONST N=5;
VAR A: ARRAY [5..8] OF INTEGER;
    T: ARRAY [0..N] OF REAL;
```

A und T werden durch diese Variablendeklaration als Arrays vereinbart. Dabei wird sowohl die Menge der zulässigen Indizes (der Indextyp) als auch der Typ der Elemente des Arrays (der Elementtyp) angegeben.

Die Variable A ist also ein Array, das aus Zahlen vom Typ INTEGER besteht. Zulässige Indizes für das Array A sind ganze Zahlen zwischen 5 und 8. Die Struktur von A entspricht einer Tabelle mit vier Elementen. Jedes Element besitzt einen Index und enthält eine ganze Zahl:

5:	0
6:	66
7:	77
8:	3

Bild 11: *Struktur des Arrays A*

Die folgenden Zuweisungen füllen das Array A mit Werten:

```
A[5]:=0; A[6]:=66; A[7]:= 77; A[4+4]:=3
```

Die letzte Zuweisung zeigt, daß man den Index eines Elementes auch als Ergebnis eines Ausdruckes angeben kann. Dabei muß natürlich der Typ des Ausdruckes mit dem Indextyp des Arrays übereinstimmen.

Die For-Anweisung wird am häufigsten im Zusammenhang mit Arrays benutzt. Man läßt die Laufvariable den Indexbereich überstreichen und kann so in einer Wiederholung eine Operation auf alle Elemente des Arrays anwenden:

```
FOR I:= 5 TO 8 DO
  WRITELN("ELEMENT",I:3," ENTHAELT DEN WERT ",A[I])
```

In Pascal müssen die Indexgrenzen in der Deklaration durch Konstanten gegeben sein. Somit muß bereits bei der Übersetzung die Größe des Arrays festgelegt werden. Um diese Einschränkung etwas zu mildern, definiert man die Indexgrenzen mit Konstanten, wie dies im Beispiel für das Array T geschehen ist. Natürlich müssen diese Konstanten auch im Anweisungsteil z.B als Grenze für For-Anweisungen benutzt werden. Stellt sich heraus, daß die Grenze zu groß oder zu klein gewählt wurde, muß man nur die Konstante im Vereinbarungsteil anpassen und das Programm neu übersetzen.

In den bisherigen Beispielen hatten wir als Indextyp immer einen Teilbereich der ganzen Zahlen betrachtet. In einzelnen Fällen kann es jedoch auch sinnvoll sein, andere Typen als Indizes zu vereinbaren. Außer dem Typ REAL kann man alle einfachen Typen verwenden:

```
ARRAY [BOOLEAN] OF CHAR;
ARRAY [CHAR] OF INTEGER;
```

Für das letzte Beispiel wollen wir noch ein vollständiges Programm betrachten. Es soll ein Text eingelesen und die Häufigkeit aller Buchstaben gezählt werden. Die Texteingabe wird durch die Eingabe eines beliebigen Sonderzeichens beendet.

```
PROGRAM HAEUFIGKEIT (INPUT, OUTPUT);
 CONST SPACE=" ";
 VAR C: CHAR;
     H: ARRAY ["A".."Z"] OF INTEGER; (* Zählarray *)
BEGIN
  FOR C:= "A" TO "Z" DO H[C]:= 0; (* Zähler löschen *)
  READ(C);
  WHILE (C>="A") AND (C<="Z") OR (C=SPACE) DO
    BEGIN
      IF C<>SPACE THEN H[C]:= H[C]+1;
```

```
      READ(C)
    END;
  FOR C:= "A" TO "Z" DO
    WRITE(C:2, H[C]:2," !");
END.
```

Jetzt werden wir noch einige typische Algorithmen in Pascal vorstellen, die auf Arrays operieren. Dabei wird die folgende Deklaration vorausgesetzt:

```
PROGRAM FELD (INPUT, OUTPUT);
  CONST UG=1; OG=10;     (* Feldgröße *)
  VAR I,J,K: INTEGER;
      A: ARRAY [UG..OG] OF INTEGER;
      SUM, MAX, W: INTEGER;
```

Eine Operation auf allen Elementen wird mit der For-Anweisung programmiert:

```
SUM:= 0;
FOR I:= UG TO OG DO SUM:= SUM + A[I];
```

Um den größten Wert in einem Array zu finden, bestimmt man schrittweise das Maximum der ersten i Zahlen, indem man das Maximum der ersten i-1 Zahlen mit dem i. Element vergleicht:

```
MAX := A[UG];
FOR I:= UG+1 TO OG DO
  IF A[I]>MAX THEN MAX:= A[I]
```

Selbst wenn UG=OG ist, arbeitet dieses Programm korrekt, da dann die For-Schleife wegen UG+1>OG nicht ausgeführt wird. Mit diesem Suchalgorithmus kann man auch ein Array sortieren: Man bestimmt das Maximum des Arrays und vertauscht es mit dem letzten Element im Array. Anschließend wiederholt man diese Prozedur für alle Elemente ohne das Maximum und erhält dadurch den zweitkleinsten Wert. Wiederholt man dieses Verfahren bis zum kleinsten Element des Arrays, so ist das Array schließlich sortiert.

```
FOR J:= OG DOWNTO UG+1 DO
  BEGIN
   (* Bestimme K, den Index des Maximums *)
   (* im Array A von UG bis J            *)
   (* Vertausche A[J] mit A[K]           *)
  END
```

Die in Kommentarklammern angegebenen Operationen hatten wir bereits früher programmiert (s.o), so daß wir das Programm vollständig angeben können:

```
FOR J:= OG DOWNTO UG+1 DO
  BEGIN
    MAX := A[UG]; K:= UG;
    FOR I:= UG+1 TO J DO
      IF A[I]>MAX THEN
        BEGIN K:=I; MAX:= A[I] END;
    A[K]:= A[J]; A[J]:= MAX
  END
```

Ist Ihnen die genaue Funktion dieses Programmes noch nicht ganz klar, sollten Sie mit Stift und Papier das Array mit den Werten

```
8 9 1 3 4 2 5
```

nach dem angegebenen Algorithmus sortieren. Dann werden Sie auch verstehen, warum dieser Algorithmus *Sortieren durch Auswahl* heißt. In Abschnitt 2.11.5 werden wir ein anderes Sortierverfahren kennenlernen, das für große Arrays wesentlich effizienter ist.

Eine weitere elementare Operation mit Arrays besteht darin, einen vorgegebenen Wert im Array zu suchen. Das Ergebnis der Suche soll der Index des Wertes im Array sein. Offensichtlich ist hier die For-Anweisung ungeeignet. Ein erster Lösungsansatz wäre folgende Schleife:

```
(*$R+  *)
I:= UG;
WHILE (I<=OG) AND (A[I]<>W) DO I:=I+1;
IF I>OG THEN
  WRITELN("NICHT GEFUNDEN")
ELSE
  WRITELN("INDEX",I);
```

Jedoch bewirkt diese Schleife einen Zugriff auf das nicht existierende Element A[OG+1], falls der gesuchte Wert W nicht im Array A enthalten ist: Im letzten Durchlauf der Schleife ist I=OG+1. Anschließend wird die Bedingung der While-Anweisung ausgewertet. Zwar ist bereits das Ergebnis des ersten Teilausdruckes (I<=OG) FALSE, dennoch wird in Pascal ein boolescher Ausdruck immer vollständig ausgewertet. Deshalb wird bei der Berechnung des zweiten Teilausdruckes auf das Element A[OG+1] zugegriffen.

Dieser verbotene Zugriff wird normalerweise in Pascal 1.4 nicht erkannt, da Indizes nicht auf die Grenzen in der Deklaration überprüft werden. Der Kommentar in der ersten Zeile des Beispiels schaltet jedoch eine Option des Compilers ein, die unter anderem die Indexgrenzen bei jedem Array-Zugriff prüft, so daß beim Programmablauf eine Fehlermeldung erzeugt wird:

```
VALUE OUT OF BOUNDS: 11  1  10
 ERROR AT xxxx
```

Das heißt, es wurde versucht, das 11. Element anzusprechen, obwohl in der Deklaration 1 und 10 als Indexgrenzen vereinbart wurden. (Näheres siehe Dokumentation in Kapitel 4.)

Die obige Suche muß also umformuliert werden:

```
I:= UG-1;
REPEAT I:=I+1 UNTIL (A[I]=W) OR (I=OG)
```

Eine intelligentere Version der Suche vermeidet die ständige Prüfung auf das Ende des Arrays. Der *Trick* besteht darin, den gesuchten Wert am Ende des Arrays als **Marke** zu speichern, so daß spätestens dort die Suche abbricht. Dazu müssen wir aber das Array um eine Position erweitern:

```
PROGRAM SUCHE (INPUT, OUTPUT);
  CONST UG=1; OG=10; OG1=11 (* OG+1 *)
  VAR A: ARRAY [UG..OG1] OF INTEGER;
...
I:=UG; A[OG1]:W;
WHILE A[I]<>W DO I:=I+1;
IF I=OG1 THEN WRITELN("WERT NICHT GEFUNDEN")
```

Damit beschließen wir die Behandlung der Algorithmen auf Arrays. Sicher werden Sie den einen oder anderen Hinweis für eigene Pascal-Programme verwenden können. Jeder Leser, der nicht gerne jedes Mal das Rad neu erfinden will, sei auf die Standardliteratur zum Thema Strukturierte Programmierung verwiesen. Besonders nützlich ist das Buch 2 (siehe Anhang E), in dem alle Algorithmen in Form von Pascal-Programmen vorgestellt und ausführlich hergeleitet werden.

Bisher wurden nur einzelne Elemente eines Arrays verändert. Möchte man z.B. den Inhalt eines Feldes A in ein Feld B desselben Typs übertragen, so könnte dies wie folgt geschehen:

```
FOR I:= UG TO OG DO B[I]:= A[I]
```

In Pascal geht dies aber auch eleganter (und wesentlich schneller) mit dem Befehl B:=A. Voraussetzung hierfür ist, daß A und B denselben Typ besitzen. Wann zwei Variablen denselben Typ besitzen, wird im Abschnitt 2.10 genau erklärt.

2.9.2 Strings

Für die Praxis ist ein spezieller Typ von Arrays interessant. Arrays mit Elementen vom Typ CHAR lassen sich als Strings (also Zeichenketten) interpretieren:

```
VAR S1,S2: ARRAY [1.. 8] OF CHAR;
    S3   : ARRAY [1..15] OF CHAR;
```

Diese Arrays bestehen also aus acht beziehungsweise fünfzehn Zeichen. Bereits im Abschnitt 2.1 wurden Stringkonstanten beschrieben. Eine Stringkonstante mit N Zeichen besitzt implizit den Typ

```
ARRAY [1..N] OF CHAR;
```

Pascal ist bei der Behandlung von Strings sehr restriktiv. Da Strings Arrays sind, besitzen sie eine konstante Größe. Zuweisungen sind nur zwischen Strings gleicher Länge erlaubt:

```
S1:= S2                          (korrekt)
S1:="12345678"  S3:="ALPHA    "  (korrekt)
S1:= S3         S3:="ALPHA"      (falsch)
```

Andererseits sind zusätzliche Operationen auf Strings definiert: Strings gleicher Länge können mit =, <, > verglichen werden. Das Ergebnis des Vergleichs hängt vom zugrundeliegenden Zeichensatz ab (s.a. Abschnitt 2.6.4).

```
"OTTO " < "OTTO2"
"EMIL"  < "ERNA"
"Emil"  > "EMIL"    (!)
```

Die folgenden Vergleiche sind wegen der unterschiedlichen Länge der Strings nicht erlaubt:

```
IF "EGON" = "EGON " THEN ...
IF S1 < S3 THEN ...
```

Schließlich kann ein String auch als Parameter in Write-Anweisungen auftreten:

```
WRITE  ("ICH HEISSE ",S1);
WRITELN("UND NICHT ",S3:20)
```

Die Eingabe von Strings mit nur einem Befehl ist im Standard nicht vorgesehen. Viele Pascal-Implementierungen haben einen großen Satz an speziellen String-Befehlen. Sie ermöglichen auch die Definition von Strings,

die eine variable Länge besitzen. Da sich jedoch auf diesem Gebiet noch kein allgemein akzeptierter Standard herauskristallisiert hat, sind solche Routinen nicht in Pascal 1.4 aufgenommen worden.

2.9.3 Mehrdimensionale Arrays

Der Elementtyp eines Arrays ist beliebig. Deshalb kann auch ein Array aus Arrays gebildet werden:

```
CONST N=3; M=4;
VAR X: ARRAY [1..N] OF
        ARRAY [1..M] OF INTEGER;
```

Eine solche zweidimensionale Datenstruktur bezeichnet man in der Mathematik als **Matrix**. Man kann sich X als eine zweidimensionale Tabelle mit ganzen Zahlen vorstellen:

	1:	2:	3:	4:
1:	1	2	3	4
2:	5	6	7	8
3:	9	10	11	12

Bild 12: *Matrix X*

Üblicherweise wird die obige Deklaration folgendermaßen abgekürzt:

```
VAR X: ARRAY[1..N,1..M] OF INTEGER;
```

Elemente einer solchen Struktur spricht man durch zwei Indizes an:

```
X[1] [2]:= X[2] [1]   oder abgekürzt
X[1,2]   := X[2,1]
```

Es ist eine reine Konvention, bei solchen Matrizen den ersten Index als Zeile und den zweiten Index als Spalte zu bezeichnen. Am Beispiel der Ein- und Ausgabe von Matrizen können Sie Ihre Kenntnisse über die Standardprozeduren READ und WRITE wieder auffrischen:

Zunächst soll die Matrix eingelesen werden. Die Eingabe soll so erfolgen, daß der Benutzer die Elemente zeilenweise eingibt:

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
```

Nach dieser Eingabe soll also gelten: X[1,2] = 2 X[3,1]=9. Offensichtlich brauchen wir zwei geschachtelte For-Anweisungen. Eine Laufvariable (i) indiziert die Zeilen, die andere (j) durchläuft in jeder Zeile die Spalten.

```
FOR I:= 1 TO N DO
  BEGIN
    FOR J:= 1 TO M DO
      READ(X[I,J]);
    READLN
  END
```

Das READLN sorgt also dafür, daß die Eingabe korrekt in verschiedenen Zeilen erfolgt. Analog erfolgt die Ausgabe der Werte in der Matrix durch WRITE und WRITELN:

```
FOR I:= 1 TO N DO
  BEGIN
    FOR J:= 1 TO M DO
      WRITE(X[I,J]:5);
    WRITELN
  END
```

Jetzt wird es Ihnen sicher leichtfallen, die obige Anweisungsfolge so zu modifizieren, daß die *transponierte* Matrix X gedruckt wird:

```
 1  5  9
 2  6 10
 3  7 11
 4  8 12
```

Im Abschnitt 2.9.1 wurde erklärt, daß man auch eine Zuweisung (A:=B) zwischen zwei Feldern durchführen kann. Nach der folgenden Deklaration

```
VAR S, T: ARRAY [1..3,1..8] OF CHAR;
```

könnte man die Matrizen S und T auch als zwei Tabellen mit jeweils drei Strings der Länge 8 betrachten. Deshalb kann man das Array mit folgenden Anweisungen vorbelegen:

```
S[1]:= "OTTO    ";
S[2]:= "ERNA    ";
S[3]:= "ANNA    ";
```

Mit der Anweisung S:=T, werden alle Zeilen von S nach T kopiert:

```
        +-        -+
        !"OTTO    "!
S = T = !"ERNA    "!
        !"ANNA    "!
        +-        -+
```

Außerdem kann man auch selektiv einzelne Zeilen ansprechen. Man gibt hinter dem Array-Bezeichner nur den Zeilenindex an:

```
WRITELN(S[2])      druckt   ERNA
S[3]:=S[2]         überschreibt ANNA mit ERNA
```

Solche *Block-Zuweisungen* sind in Pascal bei jedem zusammengesetzten Typ möglich. Man gibt beim Variablenbezeichner jeweils die Indizes nur bis zu der Dimension ein, die komplett verändert werden soll:

```
VAR H1,H2: ARRAY [0..20] OF
             ARRAY [1..4] OF
               ARRAY [1..10] OF INTEGER
```

Mit etwas Phantasie kann man in dieser Datenstruktur zwei Hochhäuser mit 20 Stockwerken (einschließlich Erdgeschoß) erkennen. In jedem Stockwerk gibt es vier Flure. Jeder Flur besitzt die Zimmernummern 1 bis 10. Für jedes Zimmer wird die Anzahl der Personen gespeichert. H1 und H2 sind also dreidimensional. Mit folgenden Zuweisungen können wir einige *Umzüge* vollziehen:

```
H1[0,1,2] :=H1[1,2,3]  H[1,2,3]:= 0
```

Hier zieht also eine Familie (?) aus dem 1. Stock, 2. Flur in die Nachbarwohnung um. Wenn die Bewohner des 3. Flurs im 2. Stock die Wohnungen im 4. Flur im Erdgeschoß übernehmen sollen, wird das in der Zimmerbuchführung wie folgt notiert:

```
H1[0,4]:=H1[2,3]
```

Größere Unruhe wird wohl die folgende Aktion zur Folge haben:

```
H1:=H2
```

Aufgaben

1. Schreiben Sie ein Programm, das in einem aufsteigend sortierten Array A ganzer Zahlen nach einer vorgegebenen Zahl W sucht. Dabei soll die Methode der binären Suche verwendet werden: Um im Teilarray von L bis R das Element W zu finden, vergleicht man W mit dem Wert an der mittleren Position M=(L+R) DIV 2. Je nachdem, ob W kleiner oder größer als A[M] ist, wählt man M als neue linke oder rechte Array-Grenze.

```
L:=UG; R:=OG;
REPEAT
  M:=(L+R) DIV 2;
  IF W<A[M] THEN ...  ELSE
  IF W>A[M] THEN ...
  ELSE...
UNTIL ...
```

   Sollten Sie Probleme bei der Formulierung des Abbruchkriteriums haben, können Sie eine boolesche Variable GEFUNDEN verwenden. Wenn Sie schon Programmiererfahrung besitzen, sollten Sie versuchen, in der Schleife mit nur einem Vergleich zwischen W und A[M] auszukommen. (Es geht!)

2. Schreiben Sie ein Programm, das das Pascalsche Dreieck druckt. Es enthält die Binomialkoeffizienten n über k, die man z.B zur Berechnung von (a+b)↑n benötigt:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
```

   Dabei steht in der n. Zeile an der k. Position die Summe der Zahlen in der n-1. Zeile an der k-1. und k. Position. (Genaueres finden Sie in jedem Lexikon.)

3. Schreiben Sie ein Programm, das ein Array reeller Zahlen verwaltet. Zu Anfang ist das Array unbelegt. Dann sollen Werte vom Benutzer eingegeben werden, die direkt bei der Eingabe so in das Array eingefügt werden, daß die Zahlen aufsteigend sortiert bleiben.

```
LEN:=0;
REPEAT
  Zahl einlesen;
  Zahl einfügen;
  LEN:=LEN+1
UNTIL LEN=Feldgröße
```

4. Schreiben Sie ein Programm, das einen Text zeilenweise von der Tastatur einliest (Sonderzeichen am Ende). Dieser Text soll dann im Blocksatz auf eine Spaltenbreite von N Zeichen verteilt ausgedruckt werden:

```
Dies   ist   ein
Beispiel für den
Blocksatz    mit
wenigen Spalten.
```

Sie müssen also zunächst eine Verteilung der Wörter über die Zeilen vornehmen. Anschließend können Sie die verbleibenden Leerzeichen am rechten Rand auf die Zwischenräume zwischen den Wörtern verteilen. Normalerweise verteilt man die Leerzeichen abwechselnd je eine Zeile von rechts und eine Zeile von links, um ein ausgeglichenes Schriftbild zu erhalten.

5. Schreiben Sie ein Programm, das eine reelle Zahl formatiert ausdruckt. Dabei sollen insgesamt N Zeichen gedruckt werden. M gibt die Anzahl der Nachkommastellen an:

```
X=12.34     N=10     M=2     .....12.34
X=400       N=8      M=2     ..400.00
X=9.8765    N=6      M=0     .....9
X=-32.40    N=7      M=3     -32.400
```

6. Erstellen Sie ein Programm, das in einem String S1 einen String S2 mit N Zeichen Länge sucht. Die gefundene Position soll in der Variablen M gespeichert werden.

```
S1:="Dies ist ein Testtext"
S2:="tt                    "; N=2   ergibt M=17
S2:="                      "; N=1   ergibt M=5
S2:="Egon                  "; N=4   ergibt M=0
```

Sie können das Programm erweitern, indem Sie auch einen *Joker* zulassen:

```
S2:="est??xt               "; N=7   ergibt M=15
```

```
S2:="est??xt              "; N=7   ergibt M=15
```

7. Für eine Matrix mit drei Zeilen und vier Spalten sollen folgende Werte berechnet werden:

   - Maximale Zeilensumme
   - Maximale Spaltensumme
   - Das betragsgrößte Element in der Matrix
   - Die Summe der Werte in jeder Diagonalen von links unten nach rechts oben
   - Die Summe der Werte in jeder Diagonalen von links oben nach rechts unten.

8. Programmieren Sie das Sortierverfahren **Bubblesort**. Um ein Array von N Werten zu sortieren, wird das Array (N-1)mal durchlaufen. In jedem Durchlauf werden jeweils zwei benachbarte Elemente verglichen und ausgetauscht, falls das zweite Element kleiner als das erste Element ist:

```
FOR I:= 2 TO N DO
  FOR J:= ........... DO
    IF A[J+1]<A[J] THEN
     BEGIN
       H:=A[J]; A[J]:=A[J+1]; A[J+1]:= H
     END;
```

   Bestimmen Sie die Grenzen für die For-Anweisung, indem Sie sich zunächst die Funktionsweise mit der Anweisung FOR J:= 1 TO N-1 klarmachen, und anschließend überlegen, bis zu welcher Grenze das Array im I-ten Durchlauf bereits sortiert ist.

2.10 Deklaration von Typen

In der Variablendeklaration wird der Typ als eine konstante Eigenschaft der Variablen festgelegt. Bisher kennen Sie die Standardtypen mit den vordefinierten Bezeichnern INTEGER, REAL, CHAR und BOOLEAN. Im Abschnitt 2.9 hatten wir bereits Array-Variablen definiert. Sie sind ein erstes Beispiel für Typen, die der Programmierer definiert. Wie Konstanten und Variablen können auch Typen Bezeichner erhalten. Dies geschieht im Typvereinbarungsteil, der im Programm zwischen dem Konstanten- und Variablenvereinbarungsteil steht:

```
TYPE GANZEZAHL = INTEGER;
     GROSSEZAHL= REAL;
     TMATRIX   = ARRAY[1..4,1..4] OF GANZEZAHL;

VAR  I: GANZEZAHL; J :INTEGER;
     MAT1,MAT2: TMATRIX;
     MAT3     : ARRAY[1..4,1..4] OF GANZEZAHL;
     MAT4     : TMATRIX;
```

Listing 9: *Typdeklaration*

Für zusammengesetzte Typen ist die Benutzung von Typbezeichnern besonders sinnvoll. Im letzten Abschnitt wäre die Struktur der Variablen H1,H2 mit der folgenden Typdeklaration sofort erkennbar:

```
TYPE BELEGUNG = INTEGER;
     FLUR     = ARRAY[1..10] OF BELEGUNG;
     STOCK    = ARRAY[1..4]  OF FLUR;
     HAUS     = ARRAY[0..20] OF STOCK;
VAR  H1,H2: HAUS;
```

Eine wichtige Aufgabe der Typdeklaration besteht darin, die Menge der Variablen in einem Programm in Klassen aufzuteilen. Zuweisungen und Operationen sind nur zwischen den Mitgliedern kompatibler Klassen möglich. Bereits bei der Übersetzung sind dadurch weitgehende Prüfungen des Programmtextes möglich.

Die folgenden Regeln legen die Typkompatibilität in Pascal fest:

1. Zwei Variablen eines zusammengesetzten Typs (Array, Record, Menge und File) sind nur dann zuweisungskompatibel, wenn sie in **einer** Variablendeklaration oder mit demselben Typbezeichner definiert wurden.

2. Strings (Werte des Typs ARRAY OF CHAR) können außerdem String-Variablen der gleichen Länge zugewiesen werden.

3. Werte eines Unterbereichstyps (siehe Abschnitt 2.12.2) können außerdem Variablen des Basistyps zugewiesen werden.

4. Ein ganzzahliger Wert kann immer einer reellen Variablen zugewiesen werden.

Die erste Regel soll am Beispiel der Deklaration aus Listing 9 verdeutlicht werden:

`MAT1:= MAT2` (zulässig)

`MAT4:= MAT1` (zulässig)

`MAT3:= MAT1` (unzulässig)

`MAT4:= MAT3` (unzulässig)

Einige Compiler sind bei der Interpretation der Regeln zur Typkompatibilität etwas großzügiger. Sie erlauben Zuweisungen auch zwischen Variablen, die nur die gleiche Struktur besitzen (MAT3:=MAT1 ist dann zulässig, siehe auch Buch 3, Anhang E).

2.11 Prozeduren

Eine besonders erfolgreiche Strategie beim Programmentwurf besteht darin, das Ausgangsproblem in geeignet gewählte kleinere Teilprobleme zu zerlegen. Ziel dieser Zerlegung ist es, überschaubare Programmteile zu erhalten, die mit dem restlichen Programm nur über wohldefinierte Schnittstellen kommunizieren, so daß die Korrektheit der Programmteile ohne Kenntnis der *Umgebung* gesichert werden kann. Als Beispiel ist in Listing 10 die Grobstruktur eines Programmes für ein Brettspiel mit dem Computer angegeben.

```
Spielbrett belegen
REPEAT
  Spielstellung anzeigen
  Eingabe Spielerzug
  Spielstellung anzeigen
  Computerzug berechnen
UNTIL Spielende
```

Listing 10: *Brettspiel*

Jede der im Klartext angegebenen Teilaufgaben läßt sich logisch von den anderen trennen. Ein Beispiel für eine Schnittstelle zwischen den Teilaufgaben ist die Spielstellung: Sie darf nur bei der Eingabe des Spielerzuges und des Computerzuges verändert werden. Die Teilaufgabe *Spielstellung anzeigen* kann auf die Spielstellung nur lesend zugreifen.

Pascal unterstützt diese Modularisierung des Programmes durch das Konzept der Prozeduren und Funktionen. Die Schnittstellen werden durch Parameterlisten und die Sichtbarkeitsregeln für Bezeichner im Programmtext festgelegt.

In Pascal würde man jede der obigen Teilaufgaben durch ein separates Programmstück, eine Prozedur, definieren. Jede Prozedur erhält in der **Prozedurdeklaration** einen Namen (Bezeichner). Im Anweisungsteil wird durch die Angabe des Namens der Prozedur die Ausführung der angegebenen Prozedur veranlaßt. Diese Art von Anweisung nennt man einen **Prozeduraufruf**.

Besonders nützlich sind Prozeduren, die von verschiedenen Stellen aufgerufen werden. Dadurch, daß die Prozedur nur einmal deklariert werden muß, spart man nicht nur Schreibarbeit und Speicherplatz, sondern muß spätere Änderungen nur an einer Stelle durchführen.

Ein konkretes Beispiel soll die einfachste Form einer Prozedur vorstellen: Wir wollen den größten gemeinsamen Teiler der ganzen Zahlen A und B bestimmen. Diese Berechnung soll den GGT in der Variablen ERG ablegen. Die dazugehörige Prozedur ist in Listing 11 angegeben.

```
PROCEDURE GGT;
BEGIN
  X:=A; Y:=B;
  WHILE X<>Y DO
    IF X>Y THEN X:= X-Y
           ELSE Y:= Y-X;
  ERG:=X
END;
```

Listing 11: *Die Prozedur GGT*

Dies ist eine Prozedurdeklaration. Sie besteht aus dem Wortsymbol PROCEDURE und dem Prozedurbezeichner GGT, die zusammen den Prozedurkopf bilden. Zwischen den Wortsymbolen BEGIN und END stehen die Anweisungen der Prozedur.

Die Prozedurdeklarationen stehen am Ende des Vereinbarungsteils. Dort werden alle Prozeduren, die das Hauptprogramm benutzt, in der Form von Listing 12 aufgeführt. Im Anweisungsteil des Programmes steht an zwei Stellen der Bezeichner GGT. Dort wird also die Prozedur GGT aufgerufen. Der GGT ist bei der Rückkehr aus der Prozedur in der Variablen ERG gespeichert.

In den nächsten Abschnitten werden wir an Hand dieses Beispielprogrammes noch weitere Möglichkeiten von Pascal vorstellen, die es erlauben, Prozeduren noch flexibler einzusetzen.

```
 PROGRAM GGTTEST(OUTPUT);
  VAR A, B, ERG, X, Y: INTEGER;

PROCEDURE GGT;
BEGIN
  X:=A; Y:=B;
  WHILE X<>Y DO
    IF X>Y THEN X:= X-Y
           ELSE Y:= Y-X;
  ERG:=X
END; (* GGT *)

BEGIN (* HAUPTPROGRAMM *)
  A:=9; B:= 3; GGT;
  WRITELN(ERG);
  A:=8; B:= 3; GGT;
  WRITELN(ERG)
END.
```

Listing 12: *Prozeduraufrufe*

2.11.1 Lokalität von Bezeichnern

In Listing 12 wurden die Variablen X und Y nur innerhalb der Prozedur GGT verwendet. Diese Zugehörigkeit drückt man dadurch aus, daß die Variablen innerhalb der Prozedur deklariert werden.

```
PROGRAM GGTTEST(OUTPUT);
  VAR A, B, ERG: INTEGER;
PROCEDURE GGT;
  VAR X,Y: INTEGER;
BEGIN
  X:=A; Y:=B;
  WHILE X<>Y DO
    IF X>Y THEN X:= X-Y
           ELSE Y:= Y-X;
  ERG:=X
END; (* GGT *)
```

Listing 13: *Lokale Variablen*

Man bezeichnet X und Y als **lokale** Variablen der Prozedur GGT, da sie jetzt außerhalb der Prozedur nicht mehr sichtbar sind. D.h. eine Zuweisung

X:=Y im Hauptprogramm würde der Compiler mit der Fehlermeldung *Bezeichner nicht deklariert* quittieren.

Durch diese lokalen Deklarationen nimmt eine Prozedur die Form eines eigenständigen Programmes an. Tatsächlich sind alle Deklarationen, die im Hauptprogramm erlaubt sind, auch lokal möglich. Wenn Sie wieder einmal die Syntax-Diagramme im Anhang A zu Rate ziehen, werden Sie sehen, daß sich an den Programm- und den Prozedurkopf ein BLOCK anschließt. Das Syntax-Diagramm BLOCK enthält sowohl den Vereinbarungsteil als auch den Anweisungsteil.

Grundsätzlich versucht man, den Sichtbarkeitsbereich (scope) eines Bezeichners (Konstante, Variable etc.) möglichst klein zu halten. Diese Strategie wird manchmal auch als Geheimnisprinzip bezeichnet: Eine Prozedur *verbirgt* vor ihrer Umgebung nicht nur die Details ihres Anweisungsteils, sondern auch die lokalen Objekte.

Ein angenehmer Nebeneffekt dieses Prinzips der Lokalität ist eine Speicherplatzersparnis. Bei der Programmausführung wird erst beim Aufruf der Prozedur (GGT) Speicherplatz für die lokalen Objekte (die Variablen X und Y) reserviert. Am Ende der Ausführung der Prozedur wird dieser Speicherplatz wieder freigegeben und steht anderen Prozeduren zur Verfügung. Eine Folge dieser Speicherorganisation ist, daß bei jedem neuen Aufruf einer Prozedur alle lokalen Variablen **undefiniert** sind.

Soll eine Variable ihren Wert zwischen zwei Aufrufen beibehalten, so muß man sie außerhalb der Prozedur (**global**) deklarieren (Listing 14).

In Listing 12 waren auch X und Y globale Variablen. Gerade in großen Programmen ist die Verwendung von globalen Variablen eine schwer zu entdeckende Fehlerquelle: Hätte man im Hauptprogramm in Listing 12 die Variablen X und Y benutzt, so würden als **Seiteneffekt** bei jedem Aufruf von GGT die Werte von X und Y überschrieben werden.

```
PROGRAM GLOBALTEST(OUTPUT);
 VAR G: INTEGER;

PROCEDURE GLOBAL;
BEGIN
 G:=G+1; WRITELN("AUFRUF NUMMER",G)
END; (* GLOBAL *)

BEGIN
 G:=0; GLOBAL; GLOBAL; GLOBAL
END.
```

Listing 14: *Globale Variable*

Nachdem nun der Unterschied zwischen globalen und lokalen Objekten einer Prozedur bekannt ist, wollen wir uns mit der Schachtelung von Prozeduren beschäftigen. Da alle Arten von Deklarationen in einer Prozedur erlaubt sind, kann eine Prozedur auch eine weitere Prozedurdeklaration enthalten.

```
PROGRAM SCHACHTELUNG (OUTPUT);
  VAR A: REAL; B: REAL;

PROCEDURE AUSSEN;
  VAR A: INTEGER;
  PROCEDURE INNEN;
    VAR I: INTEGER;
  BEGIN
    WRITELN("INNEN")
  END; (* INNEN *)

BEGIN (* AUSSEN *)
  WRITELN("AUSSEN");
  INNEN; INNEN; INNEN
END; (* AUSSEN *)

BEGIN (* HAUPTPROGRAMM *)
  AUSSEN; AUSSEN
END.
```

Listing 15: *Verschachtelung*

Lokale Prozeduren (INNEN) verwendet man aus dem gleichen Grund wie lokale Variablen: Die Prozedur INNEN wird nur in der Prozedur AUSSEN benötigt. Deshalb sollte die Umgebung (in diesem Fall das Hauptprogramm) keinen Zugriff auf die lokale Prozedur besitzen.

In Listing 13 wurden außerdem Variablen in verschiedenen Schachtelungsebenen deklariert, um die folgenden Regeln über die Sichtbarkeit von Bezeichnern in Pascal zu illustrieren:

1. Ein Bezeichner ist in dem Block P, in dem er deklariert wurde, sichtbar. Außerdem ist er in jedem Block sichtbar, der von P eingeschlossen wird, solange nicht Regel 2 gilt.

2. Eine Deklaration eines Bezeichners X in einem Block macht alle Deklarationen des Bezeichners X in äußeren Blöcken unsichtbar.

3. Die Standardbezeichner sind in einem imaginären Block deklariert, der das gesamte Programm umschließt.

Durch die Regel 1 ist die Variable B im Block SCHACHTELUNG, in der Prozedur AUSSEN und auch in der Prozedur INNEN sichtbar. Nach Regel 2 überdeckt die Deklaration von A:INTEGER die Variable A:REAL. In AUSSEN und INNEN ist also nur A:INTEGER sichtbar. Durch Regel 3 sind z.B. die Standardbezeichner TRUE und FALSE im gesamten Programm sichtbar.

Nur selten werden die Standardbezeichner mit neuer Bedeutung deklariert. Man könnte aber wegen Regel 2 und 3 mit der folgenden Deklaration die vordefinierte Konstante MAXINT = 32767 ersetzen:

```
VAR MAXINT: INTEGER
```

Sollten Ihnen die Regeln etwas kompliziert erscheinen, so merken Sie sich für den Augenblick nur, daß man in einer Prozedur Bezeichner unabhängig vom übrigen Programm wählen kann.

2.11.2 Parameter

Die Version der Prozedur GGT mit lokalen Variablen (Listing 13) ist immer noch nicht optimal in Pascal formuliert. Dort werden die Eingabewerte A und B sowie das Ergebnis ERG über globale Variablen übergeben. Durch die Benutzung von Parametern wird die Prozedur universeller (Listing 16).

```
PROGRAM PARAMETER(OUTPUT);
  VAR V: INTEGER;

PROCEDURE GGT(A,B: INTEGER; VAR ERG: INTEGER);
BEGIN
  WHILE A<>B DO
    IF A>B THEN A:= A-B
           ELSE B:= B-A;
  ERG:=A
END; (* GGT *)

BEGIN (* HAUPTPROGRAMM *)
  GGT(9,3,V);
  WRITELN(V);
  GGT(17+4,3,V);
  WRITELN(V)
END.
```

Listing 16: *GGT mit Parametern*

Der Prozedurkopf von GGT wird um eine Parameterliste erweitert. Die **formalen** Parameter A, B und ERG werden wie lokale Variablen deklariert. Beim Aufruf werden **aktuelle** Parameter angegeben, die die Werte der Variablen festlegen. Beim Aufruf müssen Anzahl und Typ der aktuellen und formalen Parameter übereinstimmen. Es gibt zwei verschiedene Typen von Parametern, die beide in Listing 16 verwendet wurden.

Wertparameter: Im Beispiel sind A und B Wertparameter. Sie verhalten sich in der Prozedur GGT wie *normale* lokale Variablen. Jedoch werden sie beim Aufruf der Prozedur durch Ausdrücke als aktuelle Parameter (z.B. 17+4 oder 3) initialisiert. Die Prozedur kann jetzt die Variablen A und B beliebig benutzen und ihnen auch Werte zuweisen, ohne daß in der aufrufenden Umgebung irgendeine Änderung bewirkt würde.

Damit konnten wir im Beispiel die Hilfsvariablen X und Y einsparen. Wertparameter sind die vorherrschende Parameterart, da beliebige Ausdrücke als aktuelle Parameter auftreten können:

```
PROGRAM PARAMETERDEMO (OUTPUT);
  VAR I: INTEGER;

PROCEDURE KASTEN(L, B: INTEGER; ZEICHEN: CHAR);
  VAR I,J: INTEGER;
BEGIN
  FOR I:= 1 TO B DO
    BEGIN
      FOR J:= 1 TO L DO
        WRITE(ZEICHEN);
      WRITELN
    END
END; (* KASTEN *)

BEGIN
  FOR I:= 1 TO 10 DO KASTEN(2*I,I+2,CHR(I+ORD("A")))
END.
```

Listing 17: *Parameterdemo*

Variablenparameter: Wertparameter können keine Ergebnisse aus der Prozedur an die aufrufende Umgebung zurückliefern. In solchen Fällen benutzt man Variablenparameter. Hier ist der aktuelle Parameter immer eine Variable vom Typ des formalen Parameters. Variablenparameter werden bei der Deklaration in der Parameterliste durch Voranstellen des Wortsymbols VAR gekennzeichnet.

Die Prozedur GGT liefert das Ergebnis über den Variablenparameter ERG zurück. Deshalb muß beim Aufruf der dritte Parameter immer eine Variable vom Typ INTEGER sein (z.B. V).

Bei der Ausführung der Prozedur ändert jede Zuweisung an einen formalen Variablenparameter den Wert des zugehörigen aktuellen Parameters. Somit wird durch die Zuweisung ERG:=A der Variablen V der Wert A zugewiesen.

Der Aufruf mit Wertparametern wird auch als *call by value* bezeichnet. Den Aufruf mit Variablenparametern bezeichnet man dann als *call by reference*. Dies deutet bereits auf die Realisierung der Parameter auf dem Rechner hin: Bei Wertparametern wird der Wert des Ausdruckes in den lokalen Speicherbereich der Prozedur kopiert. Bei Variablenparametern wird nur ein Verweis (eine Adresse) auf eine globale Variable übergeben.

Viele weitere Beispiele werden Sie in den nächsten Abschnitten finden. Am Ende dieses Abschnittes soll noch betont werden, daß alle Typen (auch zusammengesetzte) als Parameter auftreten können:

```
PROGRAM VEKTORSUMME(INPUT, OUTPUT);
  CONST N=5;
  TYPE VEKTOR=ARRAY[1..N] OF REAL;
  VAR X,Y,Z: VEKTOR;
      M    : ARRAY[1..N] OF VEKTOR; (* MATRIX!*)

PROCEDURE ADD (A,B: VEKTOR; VAR C: VEKTOR);
(* C:= A+B komponentenweise*)
  VAR I: INTEGER;
BEGIN
  FOR I:= 1 TO N DO C[I]:= A[I]+B[I]
END; (* ADD *)
BEGIN
  (* X,Y vorbelegen *)
  ADD(X,Y,Z);
  M[1]:=Z; M[2]:= X;
  ADD(M[1], M[2], M[4])
END.
```

Listing 18: *Vektorsumme*

Hier werden also Vektoren von fünf Zahlen als Parameter übergeben. Aktuelle und formale Parameter müssen natürlich auch hier übereinstimmen. Dabei ist es wichtig, daß Sie einen Typ-**Bezeichner** im Prozedurkopf angeben. Verboten ist also die Parameterliste:

```
PROCEDURE ADD (A,B: ARRAY[1..N] OF REAL; VAR C: VEKTOR);
```

2.11.3 Funktionen

Neben den Variablenparametern gibt es eine weitere Möglichkeit, Resultate zurückzugeben. Diese Methode lehnt sich an die Notation von Funktionen in der Mathematik an. Dort schreibt man zum Beispiel:

```
x = GGT(7,29)
```

Der Name der Funktion repräsentiert gleichzeitig den Wert der Berechnung. In Pascal definiert man solche Prozeduren, die nur einen skalaren Wert als Ergebnis liefern, als **Funktionen**:

```
PROGRAM FUNKTION(OUTPUT);
  VAR W: INTEGER;

FUNCTION GGT(A,B: INTEGER): INTEGER;
BEGIN
  WHILE A<>B DO
    IF A>B THEN A:= A-B
           ELSE B:= B-A;
  GGT:=A
END; (* GGT *)

BEGIN (* HAUPTPROGRAMM *)
  W:=GGT(12345,25325);
  WRITELN(W,GGT(9,3));
  W:=W+GGT(234,432)
END.
```

Man ersetzt also das Wortsymbol PROCEDURE durch das Wortsymbol FUNCTION. Außerdem folgt nach der (evtl. leeren) Parameterliste und einem Doppelpunkt ein Typbezeichner, der den **Ergebnistyp** der Funktion definiert. Hier sind nur skalare Typen und Zeiger (siehe Abschnitt 2.18) erlaubt. Die Syntax-Diagramme, die Sie im Zweifelsfall zu Rate ziehen können, stehen im Anhang A unter den Namen BLOCK und PARAMETERLISTE.

Innerhalb der Funktion muß das Ergebnis durch eine Zuweisung an den Funktionsbezeichner festgelegt werden. Dies geschieht hier durch die Anweisung GGT:=A. GGT ist also nicht nur der Name der Funktion, sondern auch der Name für das Ergebnis der Funktion. Die Anweisungen im Hauptprogramm zeigen, daß man Funktionsaufrufe nur in Ausdrücken, nicht aber als einzelne Anweisungen wie Prozeduraufrufe verwenden kann.

2.11.4 Standardprozeduren

Bereits bei Ihren ersten Schritten in Pascal hatten Sie die Anweisungen WRITE und READ sowie arithmetische Funktionen wie SIN und COS verwendet.

Im Unterschied zu den Wortsymbolen BEGIN und END handelt es sich hierbei um vordefinierte Standardbezeichner für Prozeduren und Funktionen. Wie wir bereits in Abschnitt 2.11.1 festgestellt haben, kann man die Standardbezeichner durch eigene Deklarationen *verdecken*. Als Beispiel wollen wir (mit einer numerisch sehr stabilen Methode) eine explizite Deklaration der Quadratwurzelfunktion geben:

```
FUNCTION SQRT(X: REAL): REAL;
  CONST EPS= 1.0E-7; (* Genauigkeit *)
  VAR Y, Z: REAL;
BEGIN
  IF X<0 THEN
    BEGIN
      WRITELN("FEHLER IN SQRT"); HALT
    END
  ELSE
   BEGIN Y:= 2; (* Startwert Z berechnen *)
     REPEAT
       Z:= Y; Y:=Y*Y
     UNTIL Y>X;
     REPEAT     (* Iteration *)
       Y:=Z; Z:=0.5*(Y+X/Y)
     UNTIL ABS(Y-Z)<=EPS
   END;
  SQRT:= Z
END; (* SQRT *)
```

Listing 19: *Deklaration der Quadratwurzelfunktion*

Durch eine Erniedrigung der Genauigkeit EPS läßt sich bei Bedarf die Geschwindigkeit der Routine erhöhen. Eine Sonderrolle nehmen die Prozeduren READ(LN) und WRITE(LN) ein, da sie eine variable Anzahl aktueller Parameter besitzen können. Diese Eigenschaft kann man durch selbstdefinierte Prozeduren und Funktionen nicht simulieren.

2.11.5 Rekursion

In Abschnitt 2.11.1 über die Schachtelung von Prozeduren wurde erklärt, daß in einem Block jede Prozedur aufgerufen werden kann, deren Bezeichner sichtbar ist. Dies bedeutet, daß sich eine Prozedur auch **selbst** aufrufen kann. Dieser Vorgang wird **Rekursion** genannt.

Durch die Tatsache, daß bei jedem Aufruf einer Prozedur Speicherplatz für die lokalen Objekte bereitgestellt wird, werden bei rekursiven Aufrufen verschiedene *Inkarnationen* der lokalen Variablen erzeugt.

Durch diese Methode der Speicherverwaltung werden beim rekursiven Aufruf einer Prozedur P die Werte der lokalen Variablen von P nicht überschrieben. Dies läßt sich am besten mit einem einfachen Beispielprogramm verdeutlichen (Listing 20):

```
PROGRAM REKURSION(OUTPUT);

PROCEDURE REKURSIV(N: INTEGER);
  VAR LOKAL: INTEGER;
BEGIN
  LOKAL:= N;
  WRITELN(" ":N, "LOKAL=", LOKAL);
  IF N<4 THEN REKURSIV(N+1); (*<<----------*)
  WRITELN(" ":N, "LOKAL=", LOKAL);
END; (* REKURSIV *)

BEGIN
  REKURSIV(1)
END.
```

Listing 20: *Rekursion*

Die Prozedur REKURSIV speichert zunächst (zu Demonstrationszwecken) den Wert des Parameters N in einer lokalen Variablen LOKAL. Dieser Wert wird nun in zwei identischen Write-Anweisungen um N Stellen eingerückt ausgedruckt.

Die eigentlich interessante Anweisung ist mit einem Pfeil markiert: Ist nämlich der Parameter N noch nicht gleich 4, so ruft sich die Prozedur selbst auf. Als Parameter für diesen Selbstaufruf wird die Zahl N+1 verwendet.

Da jeder Aufruf der Prozedur REKURSIV seine *eigenen* Variablen N und LOKAL besitzt, wird durch diesen rekursiven Aufruf der Inhalt der Variablen N und LOKAL in der aufrufenden Umgebung **nicht** verändert. Deshalb ergibt sich folgende Ausgabe:

```
LOKAL = 1
 LOKAL = 2
  LOKAL = 3
   LOKAL = 4
   LOKAL = 4
  LOKAL = 3
 LOKAL = 2
LOKAL = 1
```

Jeweils zwei Zeilen, die gleich weit eingerückt sind, stammen von derselben *Inkarnation* der Prozedur REKURSIV.

Dieses Programm ist zwar nicht sehr sinnvoll, zeigt aber deutlich die *geschachtelten* Aufrufe:

```
REKURSIV(1) ruft
  REKURSIV(2) ruft
    REKURSIV(3) ruft
      REKURSIV(4)
```

Außerdem sehen Sie, daß diese Folge rekursiver Aufrufe irgendwann beendet werden muß. Deshalb werden rekursive Aufrufe immer durch eine Bedingung kontrolliert. Im Beispiel Listing 20 ist dies die Bedingte Anweisung IF N<4 THEN...

Schon jetzt möchte ich Sie davor warnen, rekursive Prozeduren Schritt für Schritt nachzuvollziehen, indem Sie sich die Werte aller lokalen Variablen notieren und so den Programmablauf zu analysieren versuchen. Dabei werden Sie nach wenigen Schritten schon an einem heillosen Durcheinander verzweifeln.

Vielmehr muß man rekursive Prozeduren *deklarativ* verstehen. So kann man z.B. die Prozedur REKURSIV wie folgt beschreiben:

1. Drucke N Stellen eingerückt die Zahl N.
2. Ist der Text noch nicht vier Stellen eingerückt, so drucke einen Block, der N+1 Stellen eingerückt ist.
3. Drucke N Stellen eingerückt die Zahl N.

Um Ihr *deklaratives* Verständnis zu trainieren, sollten Sie jetzt ein Blatt Papier zur Hand nehmen und die Ausgabe notieren, die Sie beim Aufruf REKURSIV2(1) der folgenden Prozedur (Listing 21) erwarten:

```
PROCEDURE REKURSIV2(N: INTEGER);
  VAR LOKAL: INTEGER;
BEGIN
  LOKAL:= N;
  WRITELN(" ":N, "LOKAL=", LOKAL);
  IF N<4 THEN REKURSIV2(N+1);
  IF N<4 THEN REKURSIV2(N+1);
  WRITELN(" ":N, "LOKAL=", LOKAL);
END; (* REKURSIV2 *)
```

Listing 21: *Rekursiv (2)*

Diese Prozedur druckt also den gesamten eingerückten Block zweimal. (Ein Tip: Die Ausgabe ist genau 30 Zeilen lang!)

Das Prinzip der Rekursion besteht darin, daß man zur Lösung einer großen Aufgabe die Lösung der Aufgabe für *kleinere* Werte benutzt. Diese etwas tautologisch anmutende Aussage soll das folgende Beispiel illustrieren:

Wir wollen alle möglichen Anordnungen (Permutationen) eines Strings von n Zeichen drucken. Der rekursive Algorithmus hierfür lautet folgendermaßen:

Ist die Länge N gleich 1, so gibt es nur diese Anordnung. Drucke diese Anordnung.

Sonst: Drucke alle Möglichkeiten, die ersten (N-1) Zeichen im String anzuordnen.

Für alle Positionen i von 1 bis N-1: Tausche das i. Zeichen mit dem letzten Zeichen. Drucke alle Möglichkeiten für diese Anordnung. Mache die Vertauschung rückgängig

Dieser Algorithmus läßt sich direkt in Pascal formulieren (siehe Listing 22).

```
PROGRAM ANORDNUNGEN(INPUT, OUTPUT);

  CONST LEN = 3;
  TYPE STRING = ARRAY [1..LEN] OF CHAR;
  VAR I: INTEGER;
      A: STRING;

  PROCEDURE ANORDNUNG(S: STRING; N: INTEGER);
  (* Drucke alle Anordnungen der ersten N Zeichen *)
  (* im String S.                                 *)
    VAR C: CHAR; I: INTEGER;
  BEGIN
    IF N=1 THEN WRITE(S," ")
    ELSE
      BEGIN
        ANORDNUNG(S, N-1);
        FOR I:= 1 TO N-1 DO
          BEGIN
            C:= S[I]; S[I]:= S[N]; S[N]:= C;
            ANORDNUNG(S,N-1);
            C:= S[I]; S[I]:= S[N]; S[N]:= C;
          END
      END
  END; (* ANORDNUNG *)
```

```
BEGIN
  FOR I:= 1 TO LEN DO READ(A[I]);
  WRITELN; ANORDNUNG(A, LEN); WRITELN
END.
```

Listing 22: *Rekursiver Algorithmus*

Für die Eingabe von ABC produziert das Programm die folgende Ausgabe:

```
ABC BAC CBA BCA ACB CAB
```

Es gibt viele Beispiele, in denen die Rekursion eine elegante Lösung des Problems erlaubt. Bevor wir zum Schluß noch ein sehr effizientes rekursives Sortierverfahren vorstellen, wollen wir noch ein abschreckendes Beispiel für die unnötige Verwendung der Rekursion betrachten.

Es soll die Zahl n! = 1 * 2 * 3 * ... * n berechnet werden. In der Mathematik wird die Fakultätsfunktion gern als eine primitiv rekursive Funktion definiert. Es gilt nämlich:

```
0! = 1  und  (n+1)! = n! * (n+1) für n>=1
```

Somit läßt sich in Pascal die Zahl n! wie folgt berechnen:

```
PROGRAM FAKULTAET(INPUT, OUTPUT);
  VAR X:INTEGER;

  FUNCTION FAK(N: INTEGER): REAL;
  BEGIN
    IF N=0 THEN FAK:= 1.0
           ELSE FAK:= FAK(N-1) * N
  END; (* FAK *)

BEGIN
  REPEAT
    READLN(X);
    WRITELN(X:3,"! =",FAK(X))
  UNTIL X=0;
END.
```

Jedoch ist die iterative Lösung

```
FAK:=1;
FOR I:=1 TO N DO
   FAK:= FAK*I
```

leichter verständlich und auch effizienter. Jeder Prozeduraufruf benötigt nämlich abgesehen von dem Speicherplatz für die lokalen Variablen auch einen gewissen Verwaltungsaufwand, der bei der For-Schleife vermieden wird.

Jetzt kommen wir zu dem angekündigten Sortieralgorithmus: Die Aufgabe besteht darin, ein Array A mit N ganzen Zahlen aufsteigend zu sortieren.

Die Idee zu einer rekursiven Lösung besteht darin, das Array in zwei Teile A[1..K] und A[K+1..N] aufzuteilen, so daß jeder Teil für sich, ohne Kenntnis der Zahlen im anderen Teil, sortiert werden kann. Diese Zerlegung und den Index K findet man mit der folgenden Strategie:

1. Man wählt einen (zufälligen) Wert X aus dem Array. Dies kann z.B. der Wert in der Mitte des Arrays sein.

2. Anschließend bestimmt man den Index K so, daß die linke Hälfte A[1..K-1] des Arrays nur Werte kleiner oder gleich X enthält, während die rechte Hälfte A[K+1..N] nur Werte größer oder gleich X enthält.

3. Sortiert man anschließend die beiden Teilarrays, die mindestens ein Element weniger als das ursprüngliche Array enthalten, so ist schließlich das gesamte Array A[1..N] sortiert.

Die Tatsache, daß in jedem Schritt die Arraygröße um mindestens ein Element sinkt, ist wichtig, damit die Rekursion auch korrekt terminiert.

Betrachten wir diese Strategie an einem Beispiel:

```
8 5 7 6 3 4 8 4
```

Da acht Elemente vorliegen, wählt man in Schritt 1 das 4. Element (X=6) im Array. Die Zerlegung in zwei Teile sieht nach dem 2. Schritt wie folgt aus:

```
4 5 4 3   6   7 8 8
```

Links und rechts von der Zahl 6 sind (in beliebiger Reihenfolge) die Zahlen kleiner und größer als 6 aufgeführt. Da es vier Zahlen kleiner als 6 gibt, wählen wir den Index K=4+1=5. Im 3. Schritt werden nun die Teilarrays getrennt sortiert:

```
3 4 4 5   6   7 8 8
```

Damit haben wir die gesamte Folge sortiert. Für Schritt 2 (die Zerlegung in zwei Teilarrays) wollen wir noch einen genaueren Algorithmus angeben:

2.1 Setze zwei Zeiger I und J auf das erste und letzte Element im Array.

2.2 Lasse I nach *rechts* wandern, bis es auf ein Element zeigt, das größer als X ist.
Lasse J nach *links* wandern, bis es auf ein Element zeigt, das kleiner als X ist.

2.3 Da A[I] und A[J] jeweils auf der *falschen* Seite stehen, tausche die beiden Elemente aus.

2.4 Wiederhole diese Schritte, bis sich beide Zeiger I und J treffen.

Die obigen Schritte beschreiben den Algorithmus **Quicksort** von C.A.R. Hoare, der in Listing 23 als Pascal-Programm formuliert ist. Um die Korrektheit für alle Belegungen des Arrays zu sichern, ist die exakte Formulierung der Bedingungen in den Schleifen nötig. Die Diskussion solcher Details und eine Berechnung der Rechenzeit finden Sie in (2).

Die Sortierung erfolgt in der rekursiven Prozedur QUICK, deren Parameter L und R den Index des ersten und letzten Elementes des zu sortierenden Arrays enthalten. In Listing 23 sind zur Verdeutlichung die Nummern der obigen Schritte als Kommentare angegeben.

```
PROGRAM SORTIEREN (INPUT, OUTPUT);
  CONST N = 16;
  VAR A: ARRAY [1..N] OF ELEMENT;

PROCEDURE ERZEUGEN;
  VAR I: INTEGER;
BEGIN
  WRITELN("UNSORTIERTE FOLGE EINGEBEN!");
  FOR I:= 1 TO N DO READ(A[I]);
  READLN
END; (* ERZEUGEN *)

PROCEDURE AUSGEBEN;
  VAR I: INTEGER;
BEGIN
  WRITELN;
  WRITELN("DIE FOLGE LAUTET");
  FOR I:= 1 TO N DO WRITE(A[I]:5);
  WRITELN
END; (* AUSGEBEN *)

PROCEDURE QUICK (L,R: INTEGER);
  VAR I, J: INTEGER;
      X, Y: INTEGER;
```

```
BEGIN
  X:=A[(L+R) DIV 2]
  I:=L; J:=R;                          (* 2.1 *)
  REPEAT
    WHILE A[I]<X DO I:=I+1;            (* 2.2 *)
    WHILE A[J]>X DO J:=J-1;            (* 2.2 *)
    IF I<=J THEN
      BEGIN                            (* 2.3 *)
        Y:= A[I]; A[I]:= A[J]; A[J]:= Y;
        I:= I+1; J:= J-1
      END
  UNTIL I>J;                           (* 2.4 *)
  IF I<R THEN QUICK(I,R);              (* 3   *)
  IF L<J THEN QUICK(L,J)               (* 3   *)
END; (* QUICK *)

BEGIN
  ERZEUGEN; AUSGEBEN;
  QUICK(1,N); AUSGEBEN
END.
```

Listing 23: *Quicksort*

Aufgaben

1. Formulieren Sie einige der Lösungen der Aufgaben früherer Abschnitte als Prozeduren oder Funktionen. Überlegen Sie, welche Parameter diese Prozeduren benötigen. Beachten Sie dabei die Unterschiede zwischen Variablen- und Wertparametern. Diese Prozeduren können Sie als Include-Files (siehe Abschnitt 4.4.6.2) auf einer Diskette speichern und später in eigenen Programmen verwenden. Beispiele:

   ```
   PROCEDURE WRITEREAL(X: REAL; N,M: INTEGER);
   (* Drucke X in ein Feld der Größe N mit   *)
   (* M Nachkommastellen.                    *)

   PROCEDURE BLOCKSATZ(VAR S: STRING; N: INTEGER;
                       RECHTS: BOOLEAN);
   (* Formatiere Textzeile in S im Blocksatz *)
   (* auf N Stellen. Falls RECHTS=TRUE werden*)
   (* Leerstellen von rechts eingefügt       *)
   ```

   Solche kommentierte Prozedurrümpfe sind eine einfache Methode, größere Programme überschaubar zu halten.

2. Formulieren Sie ein *string-package*. Diese Prozedursammlung soll die elementaren Befehle zur String-Behandlung umfassen, so daß es in anderen Programmen (z.B. als Include-File) verwendet werden kann. Da man in Pascal keine Strings variabler Länge definieren kann, hat sich folgende Darstellungsform von Strings durchgesetzt:

```
CONST MAXLEN = 40; (* Maximale Länge eines Strings*)
TYPE  STRING = ARRAY[0..MAXLEN] OF CHAR;
```

Dabei speichert man an der Position 0 im String die tatsächliche Länge des Strings (zwischen 0 und MAXLEN). Um in S: STRING fünf Sterne zu speichern, würde man folgende Zuweisung vornehmen:

```
    FOR I:= 1 TO 5 DO S[I]:= "*";
S[0]:= CHR(5); (* 5 Zeichen lang *)
```

Durch diese Speicherungsform lassen sich die String-Prozeduren relativ effizient programmieren. Ein komplettes Beispiel soll Ihnen das Prinzip verdeutlichen:

```
PROCEDURE CONCAT(S1,S2: STRING; VAR S3: STRING);
(* Verkette S1 und S2 zu S3. Überlange Strings *)
(* werden abgeschnitten                        *)
  VAR I,J: INTEGER;
BEGIN
  J:= ORD(S2[0]); (* Länge S2 *)
  I:= ORD(S1[0])+ J;
  IF I>MAXLEN THEN BEGIN J:=J+MAXLEN-I; I:=MAXLEN-END;(* evtl. abschneiden*)
  S3:= S1; (* kopiere S1 *)
  S3[0]:= I; (* Länge S3 *)
  (* S2 nach S3 kopieren:*)
  WHILE J<>0 DO
    BEGIN
     S3[I]:=S2[J]; J:=J-1; I:=I-1
    END
END; (* CONCAT *)
```

Nach diesem Schema können Sie jetzt sicher selbst die folgenden Prozeduren und Funktionen programmieren:

```
FUNCTION LENGTH(S: STRING): INTEGER;
(* Liefert die Länge des Strings S        *)

PROCEDURE DELETE(VAR S: STRING; POS, N: INTEGER);
(* Löscht N Zeichen aus S ab Position POS  *)

PROCEDURE INSERT(S: STRING; VAR T: STRING;
                 POS: INTEGER);
(* Fügt S in T ab POS ein                 *)

PROCEDURE COPY(S: STRING; VAR T: STRING;
                 POS, N: INTEGER);
(* Kopiert von S N Zeichen ab POS nach T   *)

PROCEDURE WRITESTRING(S: STRING; N: INTEGER);
(* Drucke S in ein Feld mit N Stellen      *)
```

Falls Sie noch Zeit haben, können Sie auch die Prozeduren VAL und STR definieren, die Strings in Zahlen und Zahlen in Strings konvertieren. Bei VAL sollten Sie einen Parameter definieren, der die Po-

sition eventueller Fehler markiert bzw. angibt, an welcher Position im String die Zahl endet.

3.
```
FUNCTION GENAUIGKEIT: REAL;
  VAR R: REAL;
BEGIN
  R:= 1.0;
  REPEAT
    R:= R * 0.5
  UNTIL R+1.0<1=1.0;
  GENAUIGKEIT:= R
END; (* GENAUIGKEIT *)
```

Welches Ergebnis liefert die obige Funktion? Sollte Ihnen die etwas eigenartige Bedingung der Repeat-Anweisung Schwierigkeiten bereiten, sollten Sie Abschnitt 2.6 zu Rate ziehen.

4. Das Standardbeispiel für die Anwendung rekursiver Prozeduren sind die *Türme von Hanoi*: Gegeben sind drei Stäbe und N Scheiben, die auf die Stäbe gesteckt werden können. Die Scheiben sind der Größe nach numeriert. Zu Beginn sind alle N Scheiben der Größe nach sortiert zu einem Turm auf dem 1. Stab gestapelt (siehe Bild 13).

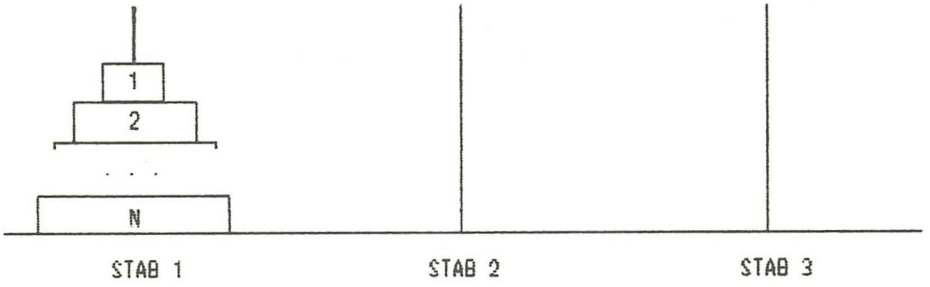


Bild 13: *Türme von Hanoi*

Die Aufgabe ist nun, den Turm in der gleichen Form auf Stab 3 aufzubauen. Dabei darf jedoch in jedem Schritt nur eine Scheibe von einem Stab zum anderen verlegt werden. Außerdem darf nie eine größere Scheibe auf einer kleineren liegen.

Schreiben Sie eine rekursive Prozedur, die einen Turm der Höhe N von 1 nach 3 verlegt. Die Lösungsidee besteht darin, daß zuerst ein Turm der Höhe N-1 von 1 auf den Hilfsstab 2 gebracht werden muß, um die Scheibe N von 1 nach 3 zu verlegen. Anschließend kann man den Turm der Höhe N-1 von 2 nach 3 bewegen und hat somit die Aufgabe gelöst.

2.12 Skalare Typen und ihre Operationen

Dieser Abschnitt setzt Abschnitt 2.6 fort. Es werden Methoden vorgestellt, um in Pascal neue einfache Typen zu deklarieren. Die so definierten Typen erlauben es, im Rechner ein möglichst exaktes Modell der Realität zu bilden.

2.12.1 Aufzählungstypen

Im Typvereinbarungsteil kann man eine Menge von Werten aufzählen und zu einem Typ zusammenfassen:

```
TYPE WOTAG=(MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG, FREITAG, SAMSTAG, SONNTAG);
FAMSTAND=(LEDIG, VERHEIRATET, GETRENNT, GESCHIEDEN, VERWITWET);
FRUCHT=(APFEL, BIRNE, ORANGE);
VAR HEUTE, MORGEN: WOTAG;
LIEBLINGSFRUCHT: FRUCHT;
```

Die Variablen HEUTE und MORGEN können nur die Werte MONTAG bis SONNTAG annehmen. WOTAG, FAMSTAND und FRUCHT bezeichnet man als Aufzählungstypen. Die Bezeichner in Klammern sind Konstanten des jeweiligen Aufzählungstyps. MONTAG ist also eine Konstante vom Typ WOTAG. Deshalb sind die folgenden Operationen nicht erlaubt:

```
HEUTE:= LIEBLINGSFRUCHT  MORGEN:= 4
IF HEUTE = LEDIG THEN...
```

Durch die Typdeklaration wird eine Ordnung auf den Konstanten definiert:

```
ORD(MONTAG)=0  ORD(DIENSTAG)=1  ... ORD(SONNTAG)=6
MONTAG<DIENSTAG  MITTWOCH>MONTAG  LEDIG<VERWITWET
IF HEUTE<=FREITAG THEN (* Werktag *)
IF LIEBLINGSFRUCHT>APFEL THEN ...
```

Die Standardprozeduren SUCC und PRED liefern zu jedem skalaren Typ den Nachfolger und Vorgänger im Wertebereich.

```
SUCC(DIENSTAG)=MITTWOCH  PRED(SONNTAG)=SAMSTAG
SUCC(APFEL)=BIRNE        PRED(VERHEIRATET)=LEDIG
```

aber auch

```
SUCC(1)=2  SUCC(-2)=-1  SUCC(FALSE)=TRUE
SUCC("A")="B"
```

Im Rechner existiert zur Laufzeit nur die kompakte Darstellung über die Ordinalwerte. Deshalb kann man die Bezeichner der Werte des Aufzählungstyps nicht direkt ausgeben:

```
WRITE("Heute ist ",HEUTE)
```
(falsch!)

Eine Ausgabe muß man explizit programmieren:

```
PROCEDURE WRITEWOTAG(TAG: WOTAG);
BEGIN
  CASE TAG OF
   MONTAG  : WRITE("MONTAG");
   DIENSTAG: WRITE("DIENSTAG");
   ...
   SONNTAG : WRITE("SONNTAG")
  END (* CASE *)
END; (* WRITEWOTAG *)
...
WRITE ("Heute ist "); WRITEWOTAG(HEUTE)
```

Aufzählungstypen erhöhen die Lesbarkeit eines Programmes enorm und sollten wo immer möglich verwendet werden. Ein etwas spezielleres Beispiel zeigt die Verwendung eines Aufzählungstyps als Indextyp: In einem Programm soll die Anzahl der Bürger jedes Familienstandes gezählt werden.

Die Idee besteht darin, ein Array von Zählern zu deklarieren, das direkt durch Werte vom Typ FAMSTAND indiziert wird. Damit kann man in einer Schleife, die alle Bürger erfaßt, mit einer einzigen Zuweisung den jeweiligen Zähler erhöhen (siehe Listing 24).

```
TYPE FAMSTAND=(LEDIG, VERHEIRATET, GETRENNT,
               GESCHIEDEN, VERWITWET);
VAR ANZAHL: ARRAY[FAMSTAND] OF INTEGER;
    STATUS: FAMSTAND;
...
ANZAHL[STATUS]:= ANZAHL[STATUS] + 1;
```

Listing 24: *Typ Familienstand*

2.12.2 Unterbereichstypen

Gibt es in einem Programm Variablen, die nur Werte aus einem Teilbereich des Wertebereichs eines Typs annehmen (oder annehmen sollen), so läßt sich diese Information bei der Deklaration einer Variablen angeben:

```
CONST N=3; M=4;
TYPE ZEILENINDEX  = 1..N;
     SPALTENINDEX = 1..M;
     DATEITYP=(SEQ, INDSEQ, REL, ERASED);
VAR M: ARRAY[ZEILENINDEX, SPALTENINDEX] OF REAL;
    I, I1: ZEILENINDEX;
    J, J1: SPALTENINDEX;
    B, BUCHSTABE: "A".."Z";
    ZIFFER:"0".."9";
    ARBEITSDATEI: SEQ..REL;
```

Man definiert sich also durch die Angabe von zwei Konstanten eines Standardtyps oder eines Aufzählungstyps einen neuen Typ, den man auch als Indextyp für Arrays verwenden kann.

Variablen dieser Unterbereichstypen nehmen nur Werte des angegebenen Intervalls an:

- Die ARBEITSDATEI darf also nicht gelöscht (ERASED) sein.
- Die Werte der Variablen I,I1 nehmen nur ganze Zahlen zwischen 1 und N an.
- Der Variablen B darf man nur Großbuchstaben zuweisen.

Jeder Unterbereichstyp grenzt den Wertebereich des dazugehörigen **Basistyps** ein. Der Basistyp des Typs ZEILENINDEX ist der Standardtyp INTEGER. Mit Variablen eines Unterbereichs sind die gleichen Operationen wie mit Variablen des Basistyps möglich.

```
I:= I1*2; A[I,J]:=4.0/ A[I-1,J+J1];
B:= CHR(68); ZIFFER:= PRED("4");
ARBEITSDATEI:= SEQ;
```

Obwohl die Einführung von Unterbereichstypen etwas mehr (Schreib-) Aufwand bei der Programmierung erfordert, ermöglicht sie eine zusätzliche Sicherheit vor unzulässigen Zuweisungen und erlaubt so eine einfachere Fehlersuche.

Wählen Sie nämlich bei der Compilation mit dem aktiven Kommentar (*$R+ *) die Option *Bereichstest ein*, so wird die Einhaltung der Intervallgrenzen geprüft. Bei allen nachfolgenden Zuweisungen, bei denen einer

Variablen eines Unterbereichstyps ein ungültiger Wert zugewiesen werden könnte, erzeugt der Compiler zusätzlichen Code, durch den bei der Laufzeit die Einhaltung der Intervallgrenzen geprüft wird:

```
I:= I1*2  READLN(BUCHSTABE)  ZIFFER:= SUCC(ZIFFER)
```

Bei den folgenden Zuweisungen ist keine Prüfung erforderlich:

```
J:= J1; I:= I1; B:= BUCHSTABE
```

Bei der Zuweisung ZIFFER:= SUCC("9") würde das Programm gestoppt und folgende Fehlermeldung erzeugt:

```
VALUE OUT OF BOUNDS: 58 48 57
```

Das bedeutet, daß der Wert SUCC("9") mit dem Ordinalwert 58 nicht im Intervall "0".."9" mit den Ordinalwerten 48 und 57 liegt. Generell werden bei dieser Fehlermeldung nur die Ordinalwerte angegeben, da z.B. für Aufzählungstypen im Objektprogramm keine Bezeichner vorhanden sind.

Zumindest in der Testphase eines Programmes ist diese Option sehr zu empfehlen, um Indizierungsfehler und Bereichsüberschreitungen zu entdecken.

Aufgaben

1. Untersuchen Sie alle im Text angegebenen Beispielprogramme in den vorangegangenen Abschnitten. Prüfen Sie, ob in den Variablendeklarationen die Möglichkeit besteht, Unterbereichstypen zu verwenden. Prädestiniert für solche Verbesserungen sind alle Variablen, die zur Indizierung verwendet werden. (Diese Variablen heißen meist I und J.)

   Compilieren Sie ein Beispielprogramm mit der Option (*$R+*), und prüfen Sie die Reaktion auf Bereichsüberschreitungen!

2. Modifizieren Sie das Programm QUICKSORT durch die Einführung von Typbezeichnern (z.B. INDEX und ITEM) im Hauptprogramm, so daß beliebige Indexbereiche und Elementtypen sortiert werden können.

   Prüfen Sie die Richtigkeit der Änderungen, indem Sie folgendes Array sortieren:

   ```
   A: ARRAY[10..20] OF CHAR;
   ```

2.13 Mengentypen

Neben dem Array gibt es in Pascal noch weitere zusammengesetzte Typen. Zu einem skalaren Typ T läßt sich z.B. ein Typ SET OF T deklarieren, der als Werte alle möglichen **Mengen** von Werten des Typs T annimmt:

```
TYPE FRUCHT=(APFEL, BIRNE, ORANGE);
     OBST = SET OF FRUCHT;
VAR LIEFERBAR, AUSVERKAUFT: OBST;
    ZAHLENMENGE = SET OF 1..30;
    KOMMANDOS   = SET OF "A".."E";
```

Die Variablen LIEFERBAR und AUSVERKAUFT können also folgende Werte annehmen:

```
[ ]  [APFEL]  [BIRNE]  [ORANGE]
[APFEL,BIRNE]  [APFEL,ORANGE]  [BIRNE,ORANGE]
[APFEL,BIRNE,ORANGE]
```

Dies sind alle möglichen Mengen, die aus den drei Früchten gebildet werden können (2 hoch 3 Möglichkeiten). Um die Operationen mit Mengen in Pascal zu verstehen, müssen Sie sich nur an den Mengenlehreunterricht erinnern und die in der Mathematik üblichen geschweiften Mengenklammern durch die eckigen Klammern in Pascal ersetzen. Praktisch alle Operationen der Mengenlehre sind auch in Pascal verfügbar (A und B sind Mengen des gleichen Typs):

A+B	bildet die Vereinigungsmenge von A und B, das sind alle Elemente, die in A oder B enthalten sind
A*B	bildet die Schnittmenge von A und B, das sind alle Elemente, die in A und B enthalten sind
A-B	bildet die Differenzmenge von A und B, das sind alle Elemente, die in A und nicht in B enthalten sind
A=B	testet die Mengen auf Gleichheit
A<=B	prüft, ob A Teilmenge von B ist
A>=B	prüft, ob A Obermenge von B ist
a IN B	prüft, ob das Element a in der Menge B enthalten ist

[ ] bildet die leere Menge

[a,b] bildet eine Menge, die aus den Elementen a und b besteht

[a..b] bildet eine Menge, die alle Werte zwischen a und b enthält.

Wie in der Mathematik ist [APFEL] nicht gleich APFEL. Das erste ist eine einelementige Menge (vom Typ OBST) und das andere ein Wert des Aufzählungstyps FRUCHT. Außerdem enthält eine Menge kein Element doppelt. Um in einem Gemüseladen Buch über die lieferbaren Früchte zu führen, könnte man z.B die folgenden Operationen verwenden:

```
LIEFERBAR:= [APFEL, BIRNE, ORANGE]
AUSVERKAUFT:= []
LIEFERBAR:= LIEFERBAR + [BIRNE]
AUSVERKAUFT:= AUSVERKAUFT + [ORANGE]
LIEFERBAR:= LIEFERBAR - [ORANGE]
IF [BIRNE,ORANGE]<= LIEFERBAR THEN ...
LIEFERBAR := LIEFERBAR *[BIRNE]
LIEFERBAR:=[APFEL..ORANGE]
```

Sie sollten sich die Mühe machen, die Bedeutung jeder einzelnen Anweisung in Worte zu fassen, um die teilweise recht komplexen Operationen zu verstehen.

Mengenoperationen werden in vielen Programmen zur Vereinfachung von Abfragen benutzt:

```
REPEAT READ(CH) UNTIL CH IN ["j","J","n","N"]
IF CH IN["0".."9"] THEN ...
IF CH IN["0".."9","A"..."Z"] THEN ...
```

Grundsätzlich beschränkt jeder Rechner die Größe einer Menge. Als Grundtyp scheidet deshalb neben dem Typ REAL auch der Typ INTEGER aus (z.B. gibt es bereits für eine Variable vom zulässigen Typ SET OF 1..30 genau 2 hoch 30 = 1073741824 verschiedene Werte!).

Pascal 1.4 erlaubt nur Mengen von Typen, die nicht mehr als 96 Werte annehmen können. Durch diese Grenze können auch Mengen von Zeichen (SET OF CHAR) dargestellt werden, die jedoch keine Grafikzeichen (ORD(CH)>=96) enthalten dürfen.

Das folgende Programm berechnet mit dem Sieb des Erasthotenes alle Primzahlen zwischen 1 und 10000. Zur Bestimmung der Primzahlen beginnt man mit einer *Tabelle* aller Zahlen im Intervall 1 bis 10000. Nun streicht man nacheinander alle Vielfachen der Zahlen 2, 3, 5, 7, 11 etc. Am Schluß

bleiben also nur die Primzahlen in der *Tabelle* stehen. Im Programm wird diese Tabelle durch eine Menge dargestellt. Sie enthält alle Zahlen, die Vielfache einer anderen Zahl sind.

Da Mengen jedoch maximal 96 Elemente enthalten dürfen, wird ein Array von Mengen benutzt. Die erste Menge enthält die Zahlen von 0 bis 95, die zweite die Zahlen zwischen 96 und 191 etc.

```
PROGRAM SIEB(INPUT,OUTPUT);
(* PRIMZAHLEN MIT DEM SIEB DES ERASTHOTENES BESTIMMEN.    *)
(* DAS ARRAY TEILBAR SIMULIERT EINE MENGE, DIE MAX        *)
(* ELEMENTE ENTHAELT. DARSTELLUNG ERFOLGT DURCH MAX96     *)
(* MENGEN DER GROESSE SETSIZE.                            *)

  CONST MAX=10000;              (* PRIMZAHLEN VON 1 BIS MAX*)
        SETSIZE=96; MAX96=105; (* = MAX DIV SETSIZE + 1   *)

  VAR TEILBAR: ARRAY [0..MAX96] OF SET OF 0..95;
      P, Z, I: INTEGER;

  FUNCTION PRIM(Z:INTEGER):BOOLEAN;
  (* PRUEFT, OB Z PRIM IST. D.H. Z IST NICHT IN DER MENGE *)
  (* DER TEILBAREN ZAHLEN.                                *)
  BEGIN
    PRIM:=NOT((Z MOD SETSIZE) IN TEILBAR[Z DIV SETSIZE])
  END; (* PRIM *)

BEGIN
  (* DIE MENGE TEILBAR IST ZU BEGINN LEER:                *)
  FOR I:=0 TO MAX96 DO TEILBAR[I]:=[];
  P:=1;
  REPEAT
    (* SUCHE NAECHSTE PRIMZAHL ALS TEILER:                *)
    REPEAT P:=P+1 UNTIL PRIM(P);

    WRITELN("STREICHE VIELFACHE VON",P:4);
    Z:=P*P;
    WHILE Z<=MAX DO
      BEGIN                             (*STREICHE Z      *)
        I:=Z DIV SETSIZE;               (*Z IST IN MENGE I*)
        TEILBAR[I]:=TEILBAR[I] + [Z MOD SETSIZE];
        Z:=Z+P
      END;
  UNTIL P*P>MAX;
  (* DRUCKE PRIMZAHLEN: *)
  FOR I:=2 TO MAX DO
    IF PRIM(I) THEN WRITE(I:6);
  WRITELN;
END.
```

Listing 25: *Sieb des Erasthotenes*

2.14 Der Datentyp Record

In einem Array werden Elemente eines einzigen Typs zu einer Datenstruktur zusammengefaßt und über einen Index angesprochen. Um Werte verschiedener Typen zu verbinden, benutzt man Records.

```
TYPE STRING=ARRAY[1..15] OF CHAR;
     KENNZEICHEN=RECORD
                   KREIS: ARRAY[1..3] OF CHAR;
                     B  : ARRAY[1..2] OF CHAR;
                    NR  : 1..9999;
                 END;
     ADRESSE    =RECORD
                   NAME,VORNAME: STRING;
                   ORT,STRASSE : STRING;
                   HAUSNR, PLZ : INTEGER
                 END;
     KRAFTFAHRZEUGSCHEIN =
                 RECORD
                   WAGEN: KENNZEICHEN;
                   WOHNORT, STANDORT: ADRESSE;
                   LEISTUNG: INTEGER
                 END;

VAR HALTER: ADRESSE;
    AUTO1, AUTO2: KENNZEICHEN;
    SCHEIN1,SCHEIN2: KRAFTFAHRZEUGSCHEIN;
```

Listing 26: *Record-Typen*

Dieses etwas ausführliche Beispiel zeigt, wie man Attribute eines realen Objektes in Variablen vom Typ Record speichert: Ein Fahrzeugkennzeichen besteht aus dem Kürzel für den Kreis, zwei Buchstaben und einer Nummer. Im Fahrzeugschein werden für ein Fahrzeug der Halter und der Standort des Fahrzeugs eingetragen. Ein Record ist also eine Art Karteikarte mit vordefinierten Feldern.

Die obigen Typen bezeichnet man auch als Verbundtypen. Auf die **Felder** einer Record-Variablen greift man durch Nennung des Variablenbezeichners und des Feldnamens getrennt durch einen Punkt zu:

```
AUTO1.KREIS:= "M  "   AUTO2.KREIS:= "F  ";
SCHEIN1.ORT:= "NEW YORK       ";
IF SCHEIN1.LEISTUNG<28 THEN ...
```

Außerdem kann man auch über mehrere Stufen auf Record-Felder zugreifen:

```
SCHEIN1.WAGEN.KREIS := AUTO1.KREIS;
SCHEIN2.STANDORT.ORT:= SCHEIN2.WOHNORT.ORT;
```

Wirklich nützlich wird das Konzept der Records durch die Tatsache, daß man Zuweisungen zwischen kompletten Records des gleichen Typs vornehmen kann:

```
AUTO1:= AUTO2; SCHEIN1.WOHNORT:= HALTER;
SCHEIN1:= SCHEIN2
```

Dadurch werden also alle Felder eines Records kopiert. In Pascal 1.4 sind auch Vergleiche zwischen Records definiert:

```
IF SCHEIN1.WAGEN=AUTO1 THEN...
```

Die Feldnamen (Selektoren), wie ORT,B und NR, sind Bezeichner, deren Sichtbarkeit auf den Record ihrer Deklaration beschränkt ist. Man könnte also durchaus ohne Namenskonflikte eine Variable KREIS deklarieren.

Eine ähnliche Bedeutung wie die For-Anweisung für Arrays besitzt die With-Anweisung (Inspektionsanweisung) für Variablen vom Typ Record. Sie vereinfacht Ausdrücke, die mit vielen Feldern eines Records arbeiten:

```
WITH SCHEIN1 DO
BEGIN
  WITH WAGEN DO
    BEGIN KREIS:="MTK"; B:="M"; NR:= 939 END;
  WITH WOHNORT DO
    BEGIN
      NAME   :="MUELLER THURGAU";
      VORNAME:="HANS PETER     "; ...
      PLZ    :=6232; HAUSNR:=4
    END;
  STANDORT:= WOHNORT; LEISTUNG:=45
END;
```

In der Anweisung, die nach dem Wortsymbol DO der With-Anweisung folgt, ist also die Angabe *Variablenbezeichner* vor dem Feldbezeichner nicht erforderlich. Geschachtelte With-Anweisungen beziehen sich aber immer auf dieselbe Record-Variable. Die folgende Schachtelung ist also verboten, da der Record AUTO1 nicht zum Record SCHEIN1 gehört

```
WITH SCHEIN1 DO
BEGIN
  WOHNORT:=HALTER;
  WITH AUTO1 DO
    B[1]:=WAGEN.B[2]
END
```

Bei vielen Compilern (auch Pascal 1.4) bringt die With-Anweisung außerdem noch Geschwindigkeitsvorteile, da alle Operationen zum Zugriff auf die Record-Variable nur einmal benötigt werden:

```
WITH SCHEIN1.WAGEN.KREIS DO
  FOR I:= 1 TO 3 DO
    ORT[I]:=" ";
```

ist also schneller als

```
FOR I:= 1 TO 3 DO
  SCHEIN1.WAGEN.KREIS[I]:=" ";
```

Im Listing 26 wurden Arrays und Records als Teile von Records deklariert. Natürlich ist es auch erlaubt, Arrays mit Records als Elementen zu benutzen:

```
VAR ZULASSUNGEN : ARRAY[1..200] OF KRAFTFAHRZEUGSCHEIN
```

Dies ist einer der Gründe für die Flexibilität der Sprache Pascal. Die Standard- und Aufzählungstypen bilden die elementaren Bausteine, mit denen man je nach Bedarf hierarchisch strukturierte zusammengesetzte Datentypen definiert.

Aufgaben

1. Schreiben Sie ein Paket mit Programmen, das mit Bruchzahlen arbeitet. Brüche sollen nicht als Zahlen vom Typ REAL dargestellt werden, sondern als Paare von ganzen Zahlen:

   ```
   TYPE BRUCH = RECORD
                  ZAEHLER: INTEGER;
                  NENNER : INTEGER
                END;
   ```

   Damit der Zahlenbereich nicht bei den einfachsten Operationen überschritten wird, sollen Zähler und Nenner immer gekürzt vorliegen (1/134 und nicht 2345/314230). Dazu können Sie die Funktion GGT aus Abschnitt 2.11 verwenden.

   ```
   PROCEDURE KUERZE (VAR A: BRUCH);
   PROCEDURE PLUS (A,B: BRUCH; VAR C: BRUCH);
   PROCEDURE MAL (A,B: BRUCH; VAR C: BRUCH);
   PROCEDURE KEHRWERT (A: BRUCH; VAR C: BRUCH);
   FUNCTION  GROESSER (A,B:BRUCH): BOOLEAN;
   FUNCTION  GLEICH (A,B:BRUCH): BOOLEAN;
   FUNCTION  WERT (A: BRUCH): REAL;
   (* Liefert den Wert Zähler/Nenner vom Typ REAL *)
   ```

2. Sollten Sie ab und zu mit komplexen Zahlen arbeiten (müssen), ist es vielleicht interessanter, den Typ KOMPLEX mit seinen Operationen zu implementieren.

   ```
   TYPE KOMPLEX= RECORD RE,IM: REAL END;
   ```

Als Operationen bieten sich Addition, Multiplikation, Bildung der konjugiert komplexen Zahl, Betragsfunktion sowie die Umwandlung in Polarkoordinaten an.

2.15 Variante Records

Eine in der Praxis recht häufige Eigenschaft von Datensätzen ist es, für einzelne Ausprägungen der Daten **unterschiedliche** Merkmale zu enthalten. Als Beispiel betrachte man grafische Objekte. Die Aufgabe besteht darin, ein Bild durch Zusammenstellung von grafischen Primitiven (Rechtecke, Kreise, Linien, Texte) zu beschreiben:

```
CONST MAXOBJ  = 50;
TYPE TPRIMITIV= (RECK, BLOCK, KREIS, LINIE,
                TEXT, HINTERGRUND);
     TKOORD   = RECORD X,Y:INTEGER END;
     TOBJEKT  = RECORD
                 FARBE  :(ROT, GRUEN, BLAU);
                 INTENS :(HELL, DUNKEL);
                 CASE ART: TPRIMITIV OF
                  RECK, BLOCK:
                    (RECHTSUNTEN,LINKSOBEN:
                      TKOORD);
                  KREIS:
                     (MITTE : TKOORD;
                      RADIUS: INTEGER);
                  LINIE:
                      (VON, BIS: TKOORD);
                  TEXT:
                      (POS1 : TKOORD;
                       STRNG: ARRAY[1..10] OF CHAR);
                  HINTERGRUND:()
                 END;
VAR BILD: ARRAY[1..MAXOBJ] OF TOBJEKT;
    OBJEKT: TOBJEKT;
```

Listing 27: *Record mit Varianten*

In der Deklaration von Listing 27 werden Rechtecke (RECK), ausgemalte Rechtecke (BLOCK), Kreise, Linien und Texte berücksichtigt. Alle Objekte besitzen gemeinsame Merkmale: Die Farbe, die Helligkeit (INTENS) und eine Kennzeichnung, die angibt, um welches elementare Objekt es sich handelt (ART). Diese gemeinsamen Felder werden wie in einem einfachen Record definiert.

An diesen **festen** Teil schließt sich der **variante** Teil an: Er wird durch das Wortsymbol CASE eingeleitet. Ihm folgt das sogenannte Auswahlfeld (Tagfield). In diesem speziellen Beispiel ist dies das Feld ART. Der Typ des Auswahlfeldes muß ein skalarer Typ (also nicht REAL oder STRING) sein, der durch einen Typ-**Bezeichner** angegeben wird.

Nach dem Wortsymbol OF werden die einzelnen Varianten aufgeführt, die in Abhängigkeit von dem aktuellen Wert des Auswahlfeldes (ART) gültig sind. Die jeweiligen Konstanten des Typs (TPRIMITIV) werden durch Kommata getrennt. Nach einem Doppelpunkt folgen in Klammern die Felder für diese Variante. Die Struktur dieser Feldliste entspricht genau der Syntax für die Feldliste zwischen RECORD und END. Also könnten dort geschachtelt wiederum Varianten stehen.

Alle Varianten sind durch Semikola getrennt. Bitte beachten Sie, daß der variante Teil nach dem Symbol CASE **nicht** durch ein eigenes END abgeschlossen wird. Deshalb steht in Listing 27 am Ende von TOBJEKT nur ein einzelnes END.

Nach der Zuweisung

```
OBJEKT.ART:= KREIS
```

wären also neben den Feldern FARBE, INTENS die Felder der Variante KREIS gültig, die man dann wie folgt belegen kann:

```
OBJEKT.MITTE.X:= 30;
OBJEKT.MITTE.Y:= 30;
OBJEKT.RADIUS := 15;
```

Erwähnenswert ist noch die Tatsache, daß die Feldliste zu einer Varianten leer sein kann:

```
HINTERGRUND:()
```

Ist also OBJEKT.ART=HINTERGRUND, so sind nur die festen Felder (FARBE, INTENS) relevant. Die Nennung leerer Varianten dient nur der Dokumentation der Struktur und ist syntaktisch nicht verpflichtend. Die exakte Syntax von (varianten) Records geht aus dem Syntax-Diagramm FELDLISTE im Anhang A hervor.

Es ist ein schwerer Programmierfehler, auf Varianten zuzugreifen, die nicht dem aktuellen Wert des Auswahlfeldes entsprechen:

```
WITH OBJEKT DO
  BEGIN ART:=TEXT; VON.X:= 30 END;
```

Um solche Probleme zu vermeiden, bietet sich die Verwendung der Fallunterscheidung (Case-Anweisung) an:

```
WITH OBJEKT DO
BEGIN
  FARBE:=ROT;
  INTENS:=SUCC(INTENS);
  CASE ART OF
   RECK,BLOCK: BEGIN
                READK(RECHTSUNTEN);
                READK(LINKSOBEN)
               END;
   KREIS     : BEGIN
                READK(MITTE); READLN(RADIUS)
               END;
   ...
   HINTERGRUND:
  END (* CASE *)
END
```

(READK soll eine Prozedur bezeichnen, die Koordinatenpaare einliest und als Variablenparameter zurückliefert.) Die Struktur der Fallunterscheidung spiegelt also die Struktur des varianten Records wider. Hier ist jedoch die Angabe der leeren Fallmarke erforderlich, da sonst zur Laufzeit für die Variante HINTERGRUND eine Fehlermeldung (in Pascal 1.4: NO LABEL FOR CASE) ausgegeben würde.

In einigen Fällen ist es sinnvoll, die Repräsentation der Daten im Rechner zu kennen. Deshalb soll die Speicherverteilung (in Pascal 1.4) für variante Records kurz umrissen werden: Da zu einem Zeitpunkt das Auswahlfeld nur einen Wert annehmen kann, erhalten alle Varianten denselben Speicherplatz. Die Größe eines Objektes vom Typ TOBJEKT wird durch die Größe des festen Teils plus der Größe der längsten Variante bestimmt. Damit ergibt sich die in Bild 14 skizzierte Speicherverteilung.

FARBE			
INTENS			
ART			
RECHTSUNTEN.X	MITTE.X	VON.X	POS1.X
RECHTSUNTEN.Y	MITTE.Y	VON.Y	POS1.Y
LINKSOBEN.X	RADIUS	BIS.X	STRNG(1)
LINKSOBEN.Y		BIS.Y	STRNG(2)
			STRNG(3)
			STRNG(4)
			STRNG(5)
			STRNG(6)
			STRNG(7)
			STRNG(8)
			STRNG(9)
			STRNG(10)

Bild 14: *Struktur TOBJEKT*

In diesem Fall ist TEXT die längste Variante. Alle anderen Varianten werden ebenfalls mit dieser Größe gespeichert (belegen also ungenutzten Speicherplatz). Will man sehr große Arrays mit solchen Objekten bilden, so kann es sinnvoll sein, die größte Variante zu kürzen. Eine Möglichkeit besteht darin, das Feld STRNG auszulagern und durch einen Verweis in eine Tabelle mit Strings zu ersetzen:

```
TYPE STRINGREF = 1..MAX;
VAR STRINGARRAY: ARRAY [STRINGREF] OF
                   ARRAY[1..10] OF CHAR;
```

Die Variante TEXT würde also lauten:

```
TEXT:
   (POS1 : TKOORD;
    STRNG: STRINGREF);
```

Um einen Text im Array BILD an I-ter Stelle einzufügen, speichert man zunächst den String an einer freien Position im STRINGARRAY (z.B. J-tes Element). Dem Feld STRNG im Record wird dann der Index J zugewiesen.

```
WITH BILD[I] DO
BEGIN
  FARBE:= BLAU; INTENS:= DUNKEL;
  ART  := TEXT; POS1.X:= 0; POS1.Y:=0;
  (* String eintragen: *)
```

```
  STRINGARRAY[J]:="+....+....+";
  (* Referenz notieren: *)
  STRNG:= J
END;
```

Solche Speicherplatzoptimierungen sind gerade auf Mikrocomputern erforderlich und holen den angehenden Software-Engineer allzu rasch von den abstrakten Datenmodellen auf den Boden der Realität aus Bits und Bytes zurück.

Der Vollständigkeit halber sei noch erwähnt, daß es in Pascal zulässig ist, anstelle des Auswahlfeldes nur einen Typbezeichner zu nennen. Der Programmierer muß dann aus dem Kontext herleiten, welche Variante des Records gültig ist. Diese Records ohne Tagfield werden praktisch nur zu *schmutzigen* Operationen verwendet, die normalerweise (aus gutem Grund) in Pascal verboten sind: Bei diesen Operationen nutzt man aus, daß die einzelnen Varianten denselben Speicherplatz erhalten. So kann man mit der folgenden Anweisung einer Variablen eines Aufzählungstyps einen Wert zuweisen, dessen Ordinalwert gleich 3 ist:

```
TYPE AUFZHL=(V1,V2,V3,V4);
VAR  SCHLIMM= RECORD
               CASE BOOLEAN OF
                TRUE :(I:INTEGER);
                FALSE:(V:AUFZHL)
              END;

...
SCHLIMM.I:=3 (* ==> SCHLIMM.V = V4 *)
...
```

Da solche Operationen inhärent von Eigenschaften spezieller Rechner und Compiler abhängig sind, sollten Sie diese nicht in Ihren Programmen verwenden.

Listing 27 zeigt exemplarisch, wie man in Pascal Deklarationen strukturiert: Angefangen bei den elementaren Bausteinen (Indexgrenzen, Aufzählungstypen) bildet man eine höhere Abstraktionsstufe: Man arbeitet z.B. mit Koordinaten und nicht mit Integer-Zahlen. Anschließend kann man diese abstrakteren Typen noch zu Records und Arrays (Verbunden und Feldern) zusammenfassen.

Diese Vorgehensweise (von unten nach oben, *bottom up*) ist zwingend notwendig: Da der Compiler den Text in einem Durchgang liest, muß jeder (Typ-)Bezeichner vor der ersten Anwendung bereits deklariert sein. Konkret bedeutet dies, daß die Deklaration des Typs TKOORD **vor** der Anwendung des Typs in der Deklaration von TOBJEKT erfolgen muß.

2.16 Der Datentyp File

Alle bisher behandelten Daten(typen) besitzen eine fest definierte konstante Größe. Ein Vorteil von Variablen dieser Typen ist, daß sie jederzeit im Programm direkt über ihren Namen (evtl. indiziert oder mit Feldbezeichner) angesprochen werden können. In der Definition der Sprache Pascal wird jedoch auch eine Datenstruktur angegeben, deren Größe während der Laufzeit variabel ist. Dafür ist der Zugriff auf Elemente der Struktur nur in fester Reihenfolge mit speziellen Prozeduren möglich. Diese Struktur heißt **File** und formalisiert das Konzept der sequentiellen Dateien.

Zu jedem Typ T läßt sich mit FILE OF T ein File mit dem Komponententyp T bilden. Beispiele für Deklarationen von Files sind in Listing 28 gegeben. Dabei werden Typbezeichner benutzt, die bereits in den Listings 26 und 27 definiert wurden. ADRESSBUCH ist also eine Filevariable mit Komponenten vom Typ ADRESSE.

```
TYPE (* siehe Listing 26 und Listing 27 *)
VAR ZAHLENSPEICHER = FILE OF INTEGER;
    ADRESSBUCH = FILE OF ADRESSE;
    BILDDATEI  = FILE OF TOBJEKT;
```

Listing 28: *Typ Adresse*

Das englische Wort *file* bezeichnet ursprünglich einen Ordner oder eine Sammelmappe. In der Datenverarbeitung wird file am besten mit **Datei** übersetzt. Dateien sind Folgen (gleichartiger) Daten, die meist auf externen Massenspeichern abgelegt werden. Da die Verwaltung von Dateien auf verschiedenen Rechnern sehr unterschiedlich realisiert wird, beschränkt sich Pascal auf sehr elementare Operationen mit sequentiellen Dateien. Dennoch weichen viele Implementierungen der Sprache vom nachfolgend beschriebenen Standard ab.

```
VAR F: FILE OF T;
```

deklariert eine Filevariable F. Diese Variable besteht aus einer (evtl. leeren) Folge von Komponenten des Typs T. Die Anzahl der Komponenten ist veränderlich. Der Zugriff auf die Komponenten kann nur sequentiell lesend oder sequentiell schreibend erfolgen. Zu jedem Zeitpunkt ist nur eine Komponente *sichtbar*. Sie wird mit F↑ bezeichnet. F↑ wird auch die Puffervariable des Files F genannt.

2.16.1 Sequentiell schreiben

Mit dem Prozeduraufruf REWRITE(F) wird F zum Schreiben vorbereitet. Alle Komponenten der Variablen F werden gelöscht. Nun kann man der Puffervariablen F↑ einen Wert vom Typ T zuweisen. Durch den Prozeduraufruf PUT(F) wird der Inhalt der Puffervariablen als eine neue Komponente in F aufgenommen. Jeder Aufruf PUT(F) erweitert F um eine Komponente. Die Komponenten werden in der Reihenfolge gespeichert, in der sie mit PUT erzeugt wurden.

2.16.2 Sequentiell lesen

Mit RESET(F) wird der lesende Zugriff auf F vorbereitet. Mit RESET(F) wird der Puffervariablen F↑ die erste Komponente in F als Wert zugewiesen. Dieser Wert kann nun beliebig weiterverarbeitet werden. Durch den Aufruf von GET(F) wird der Wert der nächsten Komponente von F nach F↑ übertragen. Somit kann man durch eine Folge von Aufrufen der Prozedur GET jede Komponente in F erreichen. Die Funktion EOF(F) (end of file) liefert den Wert TRUE, falls beim sequentiellen Lesen das Ende des Files erreicht wurde. Dann ist der Inhalt der Puffervariablen F↑ undefiniert. Beim Schreiben mit REWRITE und PUT ist EOF(F) immer TRUE.

Ein Wechsel zwischen Lesen und Schreiben ist in beliebiger Reihenfolge möglich. Dabei ist aber zu beachten, daß Lesezugriffe nur nach RESET und schreibende Zugriffe nur nach REWRITE möglich sind. Außerdem löscht jeder Aufruf von REWRITE alle eventuell vorher in F enthaltenen Komponenten!

Damit ergibt sich das folgende Schema für die Bearbeitung von Files (Listing 29).

```
PROGRAM FILES(INPUT, OUTPUT);
  VAR ZAHLENSPEICHER: FILE OF INTEGER;
      X: INTEGER;
BEGIN (* Zahlenspeicher füllen: *)
 REWRITE(ZAHLENSPEICHER);
 REPEAT
   READLN(X);
   ZAHLENSPEICHER↑:=X;
   PUT(ZAHLENSPEICHER)
 UNTIL X=0;
 (* Zahlenspeicher lesen: *)
 RESET(ZAHLENSPEICHER);
 WHILE NOT EOF(ZAHLENSPEICHER) DO
 BEGIN
```

```
  X:= ZAHLENSPEICHER↑;
  WRITELN(X);
  GET(ZAHLENSPEICHER)
 END
END.
```

Listing 29: *Fileoperationen*

Filetypen können wie alle anderen Typen als Teil zusammengesetzter Datentypen (Record, Array) auftreten. Gewöhnlich sind jedoch keine Files mit Komponenten erlaubt, die ebenfalls Files sind. Zuweisungen zwischen Variablen vom Typ File sind nicht erlaubt. Files dürfen nur als Variablenparameter an Prozeduren übergeben werden.

Der Datentyp File wird in Pascal 1.4 praktisch wie oben beschrieben realisiert. Einen Unterschied bilden jedoch die Anweisungen RESET und REWRITE. Das Betriebssystem des C 64 erwartet nämlich genaue Angaben über jedes File. Man muß festlegen, auf welchem Peripheriegerät (Floppy, Datasette, serielle Schnittstelle) die Speicherung der Komponenten erfolgt, welchen Namen das File auf dem Medium erhält etc.

Damit Sie sich nicht um solche systemspezifischen Details kümmern müssen, ist auf der Systemdiskette ein Quelltext als Include-Datei vorhanden, der die Prozeduren RESET und REWRITE definiert. Um Pascal-Programme zu schreiben, die RESET und REWRITE benutzen, gehen Sie folendermaßen vor:

1. Sie definieren den Filetyp, mit dem Sie die Prozeduren RESET und REWRITE aufrufen wollen, im Hauptprogramm mit dem Bezeichner TAPE. Außerdem deklarieren Sie dort die Filevariable KOMMANDO und teilen dem Compiler mit, daß er das Include-File FILE.INC lesen soll:

   ```
   TYPE TAPE = FILE OF INTEGER;
   VAR KOMMANDO: TEXT;
   (*$"FILE.INC"*)
   ```

2. Am Anfang jedes Blockes, in dem eine Filevariable deklariert wird, rufen Sie die Prozedur ALLOC mit der jeweiligen Filevariablen als Parameter auf:

   ```
   ALLOC(ZAHLENSPEICHER)
   ```

3. Am Ende jedes Blockes, in dem eine Filevariable deklariert wird, rufen Sie die Prozedur FREE mit der jeweiligen Filevariablen als Parameter auf:

```
FREE(ZAHLENSPEICHER)
```

Mit diesen zusätzlichen Deklarationen können Sie jedes beliebige Pascal-Programm, das Files verwendet, auf den C 64 übernehmen.

In Buch 2 (siehe Anhang E) sind zahlreiche Programme beschrieben, um effizient Files zu sortieren. Eine der einfachsten Methoden heißt Natürliches Mischsortieren. Die Sortierung eines Files C geschieht dabei in mehreren Durchläufen. Jeder Durchlauf gliedert sich in zwei Schritte.

1. Das File C wird Komponente für Komponente auf zwei weitere Files A und B verteilt (distribute).

2. Die beiden Files A und B werden gemischt (merge). Bei dieser Operation werden aufsteigende Teilsequenzen in A und B zu längeren Sequenzen zusammengefaßt und wieder in C gespeichert.

Diese beiden Schritte werden so lange wiederholt, bis C nur aus einer aufsteigend sortierten Sequenz besteht. In Listing 30 sind die nur in Pascal 1.4 notwendigen Erweiterungen mit Kommentaren gekennzeichnet.

```
PROGRAM MERGE(INPUT,OUTPUT);
(*DIESES PROGRAMM IST EIN BEISPIEL FUER DIE VERWENDUNG DER*)
(*ANPASSUNGSROUTINEN FUER FILES. GLEICHZEITIG WIRD EIN    *)
(*BEISPIEL FUER INCLUDE-FILES GEGEBEN.                     *)
(*BEIM UEBERSETZEN MUSS DIE DISKETTE MIT DEM INCLUDE-FILE *)
(*'FILE.INC' EINGELEGT SEIN.                               *)
(*11.11.1985                                               *)
(*QUELLE: N.WIRTH: ALGORITHMEN & DATENSTRUKTUREN KAP.2.3.2*)

 TYPE ITEM=RECORD
           KEY:INTEGER
           (* HIER KOENNEN WEITERE FELDER STEHEN *)
          END;
      TAPE=FILE OF ITEM;

 VAR KOMMANDO: TEXT;        (*<------ NUR PASCAL 1.4----- *)
     C       : TAPE;
     BUF     : ITEM;

  (*$"FILE.INC" INCLUDE-DATEI LESEN   NUR PASCAL 1.4----- *)

  PROCEDURE LIST(VAR F: TAPE);
  (*ZEIGE DEN INHALT VON F AN*)
    VAR X: ITEM;
  BEGIN RESET(F);
   WHILE NOT EOF(F) DO
     BEGIN
       X.KEY:=F↑.KEY;GET(F);
       WRITE(X.KEY:4)
     END;
```

```
 WRITELN
END;(* LIST *)

PROCEDURE NATURALMERGE;
(* SORTIERE FILE C. BENUTZT ZWEI HILFSFILES A UND B      *)
  VAR L  : INTEGER;  (* ANZAHL DER LAEUFE AUF C          *)
      EOR: BOOLEAN;  (* END OF RUN, ENDE DES LAUFS       *)
      A,B: TAPE;     (* HILFSFILES                       *)

  PROCEDURE COPY(VAR X,Y: TAPE);
  (* KOPIERE KOMPONENTE VON X NACH Y, AKTUALISIERE EOR  *)
    VAR BUF: ITEM;
  BEGIN
    BUF.KEY:=X↑.KEY; GET(X);
    Y↑.KEY:=BUF.KEY; PUT(Y);
    IF EOF(X) THEN EOR:= TRUE
              ELSE EOR:= BUF.KEY>X↑.KEY
  END;(* COPY *)

  PROCEDURE COPYRUN(VAR X,Y: TAPE);
  (* KOPIERE LAUF VON X NACH Y                          *)
  BEGIN
    REPEAT COPY(X,Y) UNTIL EOR
  END;(* COPYRUN *)

  PROCEDURE DISTRIBUTE;
  (* KOPIERE LAUEFE VON C ABWECHSELND AUF A UND B       *)
  BEGIN
    REPEAT
      COPYRUN(C,A);
      IF NOT EOF(C) THEN COPYRUN(C,B)
    UNTIL EOF(C)
  END;(* DISTRIBUTE *)

  PROCEDURE MERGE;
  (* MISCHE FILE A UND B ZU FILE C                      *)

    PROCEDURE MERGERUN;
    (* MISCHE LAEUFE VON A UND B ZU LAEUFEN AUF C       *)
    BEGIN
      REPEAT
        IF A↑.KEY<B↑.KEY THEN
         BEGIN COPY(A,C);
           IF EOR THEN COPYRUN(B,C)
         END
        ELSE
         BEGIN COPY(B,C);
           IF EOR THEN COPYRUN(A,C)
         END
      UNTIL EOR
    END;(* MERGERUN *)

   BEGIN (* MERGE *)
     REPEAT
       MERGERUN; L:=L+1
     UNTIL EOF(A) OR EOF(B);
     WHILE NOT EOF(A) DO
```

```
        BEGIN
          COPYRUN(A,C); L:=L+1
        END;
      WHILE NOT EOF(B) DO
        BEGIN
          COPYRUN(B,C); L:=L+1
        END
    END;(* MERGE *)

  BEGIN (* NATURALMERGE *)
   ALLOC(A); ALLOC(B);        (*<------ NUR PASCAL 1.4----- *)
   REPEAT
     REWRITE(A); REWRITE(B); RESET(C);
     DISTRIBUTE;
     RESET(A); RESET(B); REWRITE(C);
     L:=0; MERGE; LIST(C)
   UNTIL L=1;
   FREE(A); FREE(B)           (*<------ NUR PASCAL 1.4----- *)
  END; (* NATURALMERGE *)

BEGIN (* HAUPTPROGRAMM *)
 WRITELN("SORTIEREN EINES SEQUENTIELLEN FILES:");
 WRITELN("EINGABEZAHLEN: (0 AM ENDE)");
 OPEN(KOMMANDO,8,15,"IO");  (*<------ NUR PASCAL 1.4----- *)
 ALLOC(C);                  (*<------ NUR PASCAL 1.4----- *)
 REWRITE(C); READ(BUF.KEY);
 REPEAT
   C^.KEY:=BUF.KEY; PUT(C);
   READ(BUF.KEY)
 UNTIL BUF.KEY=0;
 LIST(C);
 NATURALMERGE;
 FREE(C)                    (*<------ NUR PASCAL 1.4----- *)
END.
```

Listing 30: *Natürliches Mischsortieren*

Am Beispiel des Programmes in Listing 30 sollen Sie auch lernen, wie man ein fremdes Pascal-Programm *liest*: Dabei beginnt man am besten am **Ende** des Listings. Dort steht das Hauptprogramm, das alle Prozeduren aufruft. In diesem Fall wird dort zunächst das File C mit Werten gefüllt, die von der Tastatur eingelesen werden. Ein wichtiges Detail sind auch die Bedingungen, die Repeat- und While-Schleifen kontrollieren. Hier wird die Schleife beendet, falls eine 0 von der Tastatur gelesen wurde. Anschließend wird die Prozedur LIST mit dem File C als Parameter aufgerufen. Bei solchen Prozeduraufrufen haben Sie zwei verschiedene Möglichkeiten, die Funktion des Programmes weiter zu analysieren: Entweder versuchen Sie, die Funktion von LIST zu entschlüsseln, oder Sie schließen aus dem Namen der Prozedur und dem Kontext auf die Bedeutung der Prozedur.

An dieser Stelle ist sicherlich einsichtig, daß die Prozedur den Inhalt des Files am Bildschirm auflistet. An diesen Prozeduraufruf schließt sich die eigentliche Sortieroperation (NATURALMERGE) an. Diese Prozedur können Sie wie das Hauptprogramm in einzelne Teilschritte zerlegen.

NATURALMERGE vollzieht die oben angegebenen Durchläufe in zwei Schritten. Wieder ist die Bedingung der Repeat-Schleife (L=1) entscheidend. Offensichtlich wird die Variable L in der Prozedur MERGE verändert. Ein Blick auf die Variablendeklaration von L (im Block NATURALMERGE) wird Ihnen jetzt etwas weiterhelfen. Den Rest des Programmes sollten Sie zur Übung selbst analysieren.

Als eine kleine Hilfe sei noch der Begriff eines **Laufes** (run) erläutert. Ein Lauf ist eine geordnete Teilsequenz in einem File.

```
(1 4 3 2 7 5 6 8)
```

enthält die folgenden vier Läufe:

```
(1 4)
(3)
(2 7)
(5 6 8)
```

Das Programm MERGE liefert für die obige Zahlenfolge im File C in zwei Durchläufen ein sortiertes File:

```
C = (1 4 3 2 7 5 6 8)   (verteile auf A und B)
  A = (1 4 2 7)
  B = (3 5 6 8)         (mische A und B zu C)
C = (1 3 4 5 6 8 2 7)   (verteile auf A und B)
  A = (1 3 4 5 6 8)
  B = (2 7)             (mische A und B zu C)
C = (1 2 3 4 5 6 7 8)
```

Natürlich können Sie mit dem Algorithmus nicht nur Dateien mit ganzen Zahlen sortieren. Eine Anpassung des Programmes erfolgt bei der Deklaration des Typs ITEM und dem Feld KEY.

Eine Erweiterung des obigen Programmes zeigt auch den Sinn zusammengesetzter Datentypen, die Files als Unterstrukturen enthalten. Im sogenannten N-Weg-Mischen werden statt der zwei Hilfsfiles (A und B) N verschiedene Files vom Typ TAPE benutzt. Um auf jedes File mit einem Index zuzugreifen, kann man z.B. das folgende Array von Files benutzen:

```
CONST N=4 (* Anzahl der Files*)
TYPE  TAPE = FILE OF ...
VAR   F: ARRAY[1..N] OF TAPE;
```

Somit kann man z.B das I-te File wie folgt mit 20 Zahlen belegen:

```
REWRITE(F[I]);
FOR J:=1 TO 20 DO
  BEGIN
    F[I]↑:= J; PUT(F[I])
  END
```

Durch die Verwendung der Routinen RESET und REWRITE wahren Sie die Kompatibilität mit Standard-Pascal. Andererseits können Sie nicht so gezielt wie in BASIC auf die Peripheriegeräte des C 64 (Floppy, Datasette, Drucker, Plotter, Modems) zugreifen. Deshalb unterstützt Pascal 1.4 wirkungsvoll das Konzept logischer Dateien im Betriebssystem des C 64. Die Benutzung von OPEN- und CLOSE-Prozeduren in Pascal wird in der Dokumentation exakt definiert. Im Abschnitt 3.1 werden zusätzlich konkrete Beispielprogramme gegeben.

Aufgaben

1. Um große Datenbestände, die auf sequentiellen Files gespeichert werden müssen, zu erweitern oder zu modifizieren, verwendet man in der kaufmännischen Datenverarbeitung folgendes Verfahren:

   Eine nach einem Schlüssel (z.B. Kontonummer) sortierte **Bestands**-Datei wird mit einer nach demselben Schlüssel sortierten **Bewegungs**-Datei zu einer (ebenfalls sortierten) neuen Bestandsdatei fortgeschrieben. Man muß also alle Änderungen, die man am Bestand vornehmen will, in der Bewegungsdatei sammeln:

```
TYPE STAMMRECORD = RECORD
                     KNUMMER: INTEGER;
                     NAME: ARRAY [1..10] OF CHAR;
                     ...
                     KONTOSTAND: REAL
                   END;
     BEWEGUNG    = RECORD
                     KNUMMER: INTEGER;
                     UMSATZART: (EIN, AUS,
                                 KONTOAUFGABE);
                     WERT: REAL;
                   END;

VAR ALT, NEU: FILE OF STAMMRECORD;
    BEW: FILE OF BEWEGUNG;
```

   Während man von der Datei ALT Daten nach NEU kopiert, prüft man, ob für die gerade bearbeitete Kontonummer Umsätze vorliegen. Diese werden dann mit dem Kontostand verbucht.

Schreiben Sie ein solches Fortschreibungsprogramm, das auch mehrere Umsätze pro Konto erlaubt. Wenn es Sie mehr motiviert, ist auch die Verwaltung von Adreß-, Schallplatten- und Buchdateien möglich.

2.17 Textfiles

Die im letzten Abschnitt vorgestellten Files besitzen Komponenten eines beliebigen skalaren oder zusammengesetzten Typs. Dadurch können effizient und kompakt alle Werte der Komponenten auf einem Hintergrundspeicher dargestellt werden. Jedoch werden die Werte als Bytefolgen gespeichert. Diese Codierung der Daten ist eine für den Menschen oder Programme in anderen Programmiersprachen ungeeignete Darstellungsform. Da zur Ein- und Ausgabe bevorzugt **Zeichenfolgen**, wie

```
Dies ist
ein Text
in drei Zeilen.
```

verwendet werden, spielen Files mit dem Komponententyp CHAR eine besondere Rolle. In Pascal existiert deshalb ein vordefinierter Typbezeichner:

```
TYPE TEXT = FILE OF CHAR;
```

Die im Programmkopf genannten Bezeichner INPUT und OUTPUT sind vordefinierte Filevariablen, die durch die folgende Deklaration definiert sind:

```
VAR INPUT, OUTPUT: TEXT;
```

Bereits im Abschnitt 2.2 wurde erwähnt, daß INPUT und OUTPUT die Standardeingabe von der Tastatur bzw. die Standardausgabe an den Bildschirm symbolisieren.

Files mit dem Grundtyp CHAR werden normalerweise nicht mit GET und PUT bearbeitet. Viel bequemer ist die Verwendung der Standardprozeduren READ(LN) und WRITE(LN). In Abschnitt 2.5 wurde nämlich nur eine Kurzform dieser Prozeduren vorgestellt. Normalerweise muß bei READ und WRITE noch ein File vom Typ TEXT (also FILE OF CHAR) als erster Parameter angegeben werden:

```
WRITE(f, Zeichen)
WRITE(f, reelle Zahl)
WRITE(f, String)
WRITELN(f)
```

Die obigen Prozeduren schreiben Zeichenfolgen auf das File f. Das Format entspricht exakt den in Abschnitt 2.5 für Bildschirmausgaben beschriebenen Konventionen. Insbesondere ist auch die Angabe einer Feldlänge möglich. Gibt man kein File als ersten Parameter an, so wird die Standardausgabe OUTPUT benutzt: WRITE(A,B,C) ist also die Abkürzung für

```
WRITE(OUTPUT,A,B,C)
```

Üblicherweise sind Textfiles zusätzlich noch in **Zeilen** strukturiert (s.a. den Text am Abschnittanfang). Im File wird deshalb am Zeilenende jeweils ein spezielles Steuerzeichen (CHR(13) in Pascal 1.4) angefügt. Damit Sie sich nicht um die Realisierung der Zeilenstrukturierung kümmern müssen, ist die Prozedur WRITELN einheitlich für alle Ausgaben auf Bildschirm, Drucker und Dateien auf der Floppy verwendbar.

Wie bei allen anderen Files auch, müssen Files mit dem Komponententyp CHAR vor solchen Schreiboperationen mit RESET zum Schreiben eröffnet werden. In diesem Kapitel wollen wir gleich die Prozeduren OPEN und CLOSE von Pascal 1.4 benutzen, da Textfiles meist auf Peripheriegeräte ausgegeben werden.

In dem in Listing 31 angegebenen Programm wird wieder das Muster für eine zeilenweise Ausgabe verwendet. Es soll eine ASCII-Tabelle auf den Drucker ausgegeben werden: Die Variable I gibt die momentan ausgegebene Zeile an. In der inneren For-Anweisung für die Variable J wird der ASCII-Code mit der Schrittweite 15 berechnet. Die Zeile wird mit WRITELN(D) beendet. Bei der Ausgabe werden die Zeichen mit Codes zwischen 0 und 31 sowie 127 und 159 nicht gedruckt, da sie am Drucker nur Steuerfunktionen besitzen.

```
PROGRAM ASCII(INPUT,OUTPUT);
 VAR I,J:INTEGER;
     D: TEXT;       (* Filevariable für den Drucker *)
BEGIN
  OPEN(D,4,0);      (* Eröffnet den Druckerkanal    *)
  FOR I:= 0 TO 15 DO
    BEGIN
      FOR J:= 0 TO 15 DO
      IF J IN [0,1,8,9] THEN
        (* ignoriere Steuerzeichen   *)
        WRITE(D,"   ")
      ELSE
```

```
      WRITE(D," ",CHR(I+16*J)," ");
    WRITELN(D)
  END;
 CLOSE(D)
END.
```

Listing 31: *ASCII*

Sollten Sie keinen Drucker besitzen, so können Sie die Ausgabe auf einem anderen Peripheriegerät vornehmen, indem Sie den Geräteparameter 4 ändern (z.B. ist 3 der Bildschirm). Hier können natürlich nicht alle möglichen Geräte besprochen werden. Details über die Wahl der Parameter entnehmen Sie am besten den jeweiligen Handbüchern.

```
OPEN(D,4,7)  (bei MPS-802 Ausgabe in Kleinschrift)
OPEN(D,3,0)          (Ausgabe auf den Bildschirm)
OPEN(D,1,2,"ASCII")  (Ausgabe auf Kassetten-File)
OPEN(D,8,3,"ASCII,S,W")
```

Die letzte Angabe erzeugt eine sequentielle Datei "ASCII" auf der Diskette. Bitte beachten Sie, daß bei Kassettenoperationen Teile des Pascal-Systems überschrieben werden. Nachdem Sie ein übersetztes Pascal-Programm, das Kassettenfiles benutzt, mit RUN gestartet haben, müssen Sie das Pascal-System neu laden.

Natürlich existiert auch für beliebige Textfiles die Möglichkeit, Daten einzulesen. Hierzu werden die entsprechenden READ(LN)-Prozeduren wie bei der Tastatureingabe benutzt:

```
READ(F, Variable)
READLN(F)
```

Dabei kann ebenfalls die Angabe des Standard-Eingabefiles INPUT als Parameter entfallen.

```
READLN(INPUT,X,Y,Z)
```

kann also zu READLN(X,Y,Z) abgekürzt werden.

Um das Ende einer Eingabezeile bei Read-Operationen zu erkennen, ist die Standardfunktion

```
EOLN(F)
```

(end of line) vorhanden. Ist beim Einlesen einer Zahl oder eines Zeichens mit READ(F,...) das letzte gelesene Zeichen ein Zeilenende-Zeichen, so

liefert die Funktion EOLN(F) den Wert TRUE. Jedoch werden Sie bei der Eingabe von Textfiles nie das Zeilenende-Zeichen (CHR(13) bei Pascal 1.4) erhalten, da dieses automatisch in ein Leerzeichen umgewandelt wird:

```
READ(F,CH); B:=EOLN(F)
```

Wird in dieser Anweisungsfolge das Zeilenende von F erreicht, so liefert CH (vom Typ CHAR) als Wert ein Leerzeichen ' '. Jedoch ist dann der Wert der booleschen Variablen B TRUE.

Bereits in Abschnitt 2.5 wurde beschrieben, daß durch den Prozeduraufruf READLN der Rest einer Bildschirmzeile überlesen wird. An dieser Stelle sind Sie in der Lage, die exakte Definition in Zusammenhang mit der Zeilenstruktur von Textfiles zu verstehen.

Die Prozedur READLN(F) läßt sich formal durch die folgende Anweisungsfolge definieren:

```
WHILE NOT EOLN(F) DO READ(F,CH);
```

Das folgende Programm demonstriert die Eingabe von Files mit READ. Die Aufgabe besteht darin, ein Eingabefile mit reellen Zahlen zu lesen und Zeilensummen auszugeben. Zu der Eingabe

```
3.142   22  -0.345   0.33
1 2 3 4
3
-8 -8 -8
```

soll also die Ausgabe

```
25.127  10  3 -24
```

erzeugt werden. Um das Zeilenende zu erkennen, muß die Funktion EOLN verwendet werden.

```
PROGRAM ZEILENSUMME(INPUT,OUTPUT);
  VAR DATEN: TEXT;

PROCEDURE ADD(VAR F:TEXT);
  VAR R, SIGMA: REAL;
BEGIN
  WHILE NOT EOF(F) DO
  BEGIN SIGMA:=0;
    REPEAT
      READ(F,R); SIGMA:= SIGMA+R
    UNTIL EOLN(F);
    WRITE(SIGMA:6)
```

```
  END
END; (* ADD *)

BEGIN
  OPEN(DATEN,8,3,"DATA,S,R");
  ADD(DATEN); WRITELN;
  CLOSE(DATEN)
END.
```

Als ein Beispiel für Programme mit Ein- und Ausgabe auf Textfiles ist ein Umwandlungsprogramm angegeben. Dieses Programm liest ein sequentielles File EINGABE (auf der Diskette mit dem Namen "TEXT") und wandelt alle Grafikzeichen mit Ordinalwerten größer als 127 in Buchstaben und Sonderzeichen um. Der umgewandelte Text wird unter dem Namen "TEXT.G" ebenfalls auf Diskette gespeichert.

```
PROGRAM KONVERT(INPUT,OUTPUT);
 VAR EINGABE, AUSGABE: TEXT;
     CH: CHAR;
BEGIN
  OPEN (EINGABE, 8, 3, "TEXT,S,R");
  OPEN (AUSGABE, 8, 4, "TEXT.G,S,W");
  WHILE NOT EOF(EINGABE) DO
  BEGIN
    READ(EINGABE, CH);
    WHILE NOT EOLN(EINGABE) DO
    BEGIN
      IF ORD(CH)>127 THEN CH:=CHR(ORD(CH)-128);
      WRITE(AUSGABE, CH); READ(EINGABE, CH)
    END;
    WRITELN(AUSGABE)
  END;
  CLOSE(EINGABE); CLOSE(AUSGABE)
END.
```

Natürlich können die Programme auch mit Files auf anderen Speichermedien (oder Bildschirm und Tastatur) arbeiten, wenn Sie die Parameter bei OPEN geeignet wählen.

Abschließend muß noch erwähnt werden, daß im Standard die Prozedur READLN formal etwas anders definiert wird: READLN(F) liest so lange Zeichen vom File F, bis F^ das erste Zeichen der nächsten Zeile enthält. Diese Definition setzt aber voraus, daß die folgende Zeile bereits vorhanden ist. Dies läßt sich zwar bei Files auf externen Speichermedien realisieren, erfordert aber bei Eingaben vom Bildschirm, daß der Benutzer bereits das erste Zeichen der nächsten Zeile eingegeben hat. Dies ist jedoch bei Dialogprogrammen (READ und WRITE im Wechsel) auf dem C 64 nicht zu realisieren.

Einige weitere Hinweise und Beispiele für Files finden sich in der Dokumentation (Kapitel 4) und bei den Tips und Tricks im Kapitel 3.

Aufgaben

1. Erstellen Sie ein Druckprogramm, das den Inhalt eines Datenfiles, wie es z.B. in der Aufgabe 1 in Abschnitt 2.16 beschrieben wurde, formatiert als Liste ausgibt. Finden Sie ein möglichst allgemein verwendbares Verfahren, um jede Seite mit einem Listenkopf (mit Seitennummer) zu drucken.

2. Gegeben ist ein Datenfile, das die Umsätze von Vertretern im Bundesgebiet für ein Jahr enthält. Das File ist nach dem Feld Postleitzahl aufsteigend sortiert. Drucken Sie eine Liste, die alle Umsätze im Bundesgebiet enthält. Außerdem sollen Zwischensummen gebildet werden, aus denen die Gesamtumsätze in jedem PLZ-Bereich (also z.B. 6000-6999) hervorgehen.

   Diese *Gruppenkontrolle* läßt sich auch mehrstufig anwenden: Innerhalb jedes PLZ-Gebietes könnte man (bei einer entsprechenden Sortierung der Ausgangsdaten) auch eine zusätzliche Aufschlüsselung nach Monatsumsätzen vornehmen. Im Programm muß man also einen Vergleich des laufenden mit dem nachfolgenden (Teil-)Schlüssel vornehmen.

```
Ersten Satz lesen
WHILE NOT Dateiende erreicht DO
BEGIN
  Vorlauf Stufe 2
  REPEAT
  Vorlauf Stufe 1
    REPEAT
      Bearbeitung Einzelposten
      Neuen Satz lesen
    UNTIL Wechsel 1
    Gruppenabschluß 1
  UNTIL Wechsel 2
  Gruppenabschluß 2
END
```

   Wechsel 1 bezeichnet also einen Wechsel des Monats, während Wechsel 2 eine Änderung des übergeordneten Gruppenkriteriums (PLZ-Bereich) bedeutet. Wechsel 1 muß natürlich auch durch Dateiende und Wechsel 2 hervorgerufen werden. Gleiches gilt für Wechsel 2. Der Vorlauf für eine Gruppe enthält das Löschen von Summenfeldern, den Druck von Überschriften etc., während der Gruppenabschluß z.B. den Druck einer Summenzeile über die Gruppe bedeutet.

2.18 Dynamische Datenstrukturen

Mit Ausnahme der Variablen vom Typ File sind alle bisher vorgestellten Strukturen (Arrays, Records, Mengen) statisch. Das heißt, sie behalten während ihrer Gültigkeit die Struktur bei, die bei der Deklaration vereinbart wurde.

In diesem Abschnitt wird beschrieben, wie man in Pascal Objekte konstruiert, die während der Programmlaufzeit nicht nur wachsen oder schrumpfen, sondern auch **dynamisch** zu Listen und beliebigen Netzen verbunden werden können. Zum Zeitpunkt der Übersetzung wird nur die Struktur der (statischen) Elemente definiert. Diese *Bausteine* besitzen meist die Struktur eines Records. Der Speicherplatz für die verschiedenen Records wird dann zur Programmlaufzeit je nach Bedarf zur Verfügung gestellt. Die Verbindung zu komplexen Strukturen geschieht über **Zeiger**, die von Record zu Record führen.

Wir wollen eine Liste von Kunden bilden. Von jedem Kunden sollen der Name und die Kundennummer gespeichert werden. Da wir nicht wissen, wie viele Kunden zu speichern sind, können wir kein Array verwenden. Andererseits wollen wir nicht ständig auf ein (langsames) Diskettenfile zugreifen. Dies ist ein typisches Beispiel für die Anwendung einer **Liste**, die durch Zeiger gebildet wird.

```
TYPE KUNDENZEIGER = ↑ KUNDE;
     KUNDE = RECORD
               NAME: ARRAY [1..10] OF CHAR;
               KNUMMER: INTEGER;
               NAECHSTER: KUNDENZEIGER
             END;
VAR  KUNDE1, KUNDENEU, LETZTERKUNDE: KUNDENZEIGER;
```

Listing 32: *Zeigertypen*

Mit der Typdeklaration aus Listing 32 definieren wir einen Typ KUNDE, der die gewünschten Informationen für jeden Kunden speichert. Der Typ KUNDENZEIGER besitzt als Werte Zeiger (pointer) auf solche Kundenrecords. Als Variablen haben wir keine Kundenrecords, sondern nur Zeigervariablen deklariert.

Zum Aufbau einer (Kunden-)Liste geht man folgendermaßen vor: Man erzeugt sich für jeden neuen Kunden einen neuen Record vom Typ KUNDE. Diese Records werden nun durch Zeiger vom Typ KUNDENZEIGER *verkettet*. Der Zeiger KUNDE1 zeigt auf den ersten Kundenrecord in der Liste. Jeder Record enthält im Feld NAECHSTER einen Zeiger auf seinen Nachfolger in der Liste.

Da jeder Kundenrecord keinen eigenen Bezeichner besitzt, kann man Kundenrecords nur durch die Angabe eines Zeigers ansprechen. Man sagt deshalb auch, daß dynamische Objekte **anonym** sind.

Um Speicherplatz für einen Kundenrecord zur Verfügung zu stellen, benutzt man die Standardprozedur NEW. Sie erzeugt irgendwo im Speicher Platz für einen Record. Um nun auf diesen Record zuzugreifen, verlangt die Prozedur eine Zeigervariable vom Typ KUNDENZEIGER als aktuellen Parameter. Dieser Zeigervariablen wird die Adresse des neuen Records vom Typ KUNDE zugewiesen:

```
NEW(KUNDENEU)
```

Über den Zeiger KUNDENEU können wir jetzt den Record vom Typ KUNDE mit Werten füllen. Da bei den Zuweisungen nicht die Zeigervariable, sondern das Objekt, auf das der Zeiger zeigt, gemeint ist, benutzt man den Pfeil ^ nach dem Bezeichner.

```
KUNDENEU↑.NAME:= "JONES     ";
KUNDENEU↑.KNUMMER:= 1111
```

Da ein Zeiger eine Referenz auf ein dynamisches Objekt darstellt, nennt man den Pfeil auch Dereferenzier-Operator.

Um nun den Zeiger KUNDE1 auf den mit NEW erzeugten Kundenrecord zu setzen, führt man eine Zuweisung zwischen Zeigern durch:

```
KUNDE1:= KUNDENEU
```

Damit Sie den Unterschied zwischen Zeigern und den durch sie referenzierten Objekten erkennen, werden wir noch einen neuen Record an die Liste hängen:

Zunächst müssen wir wieder einen neuen Record vom Typ Kunde bilden:

```
NEW(LETZTERKUNDE)
```

Dann wird der Inhalt von LETZTERKUNDE↑ initialisiert:

```
LETZTERKUNDE↑.NAME:= "JACKSON   ";
LETZTERKUNDE↑.KNUMMER:= 2222
```

Wir wollen jetzt Jackson als Nachfolger von Jones in die Liste aufnehmen. Dazu verwenden wir den Zeiger NAECHSTER im Record von Jones, auf den ja noch KUNDENEU zeigt. NAECHSTER soll auf den Record von Jackson zeigen, der durch LETZTERKUNDE referenziert wird:

```
KUNDENEU↑.NAECHSTER:= LETZTERKUNDE
```

Jetzt ist es an der Zeit, die Liste zu betrachten, die wir durch die obigen Anweisungen erzeugt haben. Eine anschauliche Darstellung von dynamischen Strukturen stellt die einzelnen Records als Kästchen dar, während Zeiger durch Pfeile symbolisiert werden, die von Record zu Record führen.

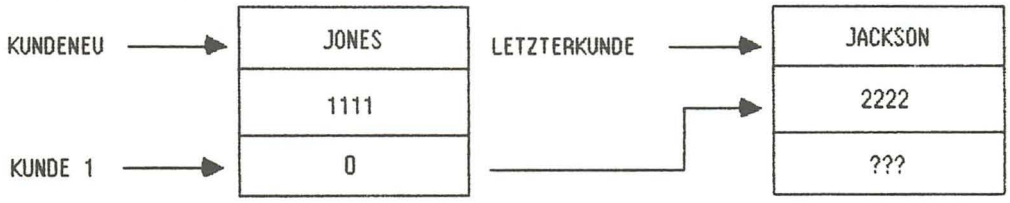


Bild 15: *Kundenliste*

Ein grundsätzliches Problem haben wir noch nicht beachtet: Was passiert mit Zeigern, die (noch) auf **kein** Element zeigen? So hat z.B. der Record LETZTERKUNDE↑ keinen Nachfolger. Eine Möglichkeit besteht darin, jeden Record um ein boolesches Feld zu erweitern, das angibt, ob ein Nachfolger existiert oder nicht. Da dieser Fall bei der Arbeit mit Zeigern ständig auftritt, ist der Wertebereich von allen Zeigertypen um den Wert NIL erweitert: Besitzt ein Zeiger P den Wert NIL, so existiert kein Objekt P↑. Deshalb füllen wir das Feld NAECHSTER bei dem Record Jackson mit NIL:

```
LETZTERKUNDE↑.NAECHSTER:= NIL
```

Bevor wir uns einigen typischen Datenstrukturen zuwenden, die mit Zeigern realisiert werden, fassen wir die Regeln für die Arbeit mit Zeiger in Pascal zusammen:

Ein Zeigertyp Z auf Objekte eines strukturierten oder unstrukturierten Typs T wird folgendermaßen deklariert:

```
TYPE Z = ↑ T
```

Soll ein Typ, der durch Zeiger angesprochen wird, selbst Zeiger enthalten, so könnten Probleme auftreten, da (wie in Abschnitt 2.15 erklärt) jeder Bezeichner vor seiner Anwendung deklariert werden muß:

```
TYPE T = RECORD
          ...
          TT: Z  <------- falsch!
         END;       (Z noch nicht bekannt)
     Z = ↑ T
```

Deshalb gibt es von dieser Regel eine Ausnahme: In der Deklaration einer Zeigervariablen kann ein Typbezeichner verwendet werden, der noch nicht deklariert wurde. Deshalb schreibt man (wie auch in Listing 32):

```
TYPE Z = ↑ T;     <------- richtig!
     T = RECORD     (T darf nach ↑ noch unbekannt sein)
          ...
          TT: Z
         END;
```

Um ein neues dynamisches Objekt vom Typ T zu erzeugen, ruft man die Prozedur NEW mit einer Variablen vom Typ Z = ↑ T auf.

Enthält eine Zeigervariable V einen Zeiger auf ein Objekt, das mit NEW erzeugt wurde, so bezeichnet V↑ dieses Objekt.

Der Wertebereich jeder Zeigervariablen V umfaßt auch den Wert NIL. Ein Zugriff auf das Element V↑ ist dann nicht zulässig.

Eine Tatsache muß noch besonders betont werden. Zwar kann ein Zeiger während der Laufzeit auf beliebige Objekte gesetzt werden, jedoch bleibt in jedem Fall die Typbindung von Pascal in Kraft. Konkret heißt dies, daß eine Zeigervariable, die mit

```
VAR V:↑ T;
```

deklariert wurde, nur auf Objekte vom Typ T zeigen kann. So ist also die folgende Anweisung nach der angegebenen Deklaration von ZI nicht zulässig:

```
VAR ZI: ↑ INTEGER;
ZI↑:= "A";
```

2.18.1 Lineare Strukturen (Listen)

Im vorangegangenen Abschnitt haben wir bereits erste Schritte zum Aufbau einer Liste von Kundenrecords gemacht. Dabei sind wir von einer intuitiven Vorstellung einer Liste ausgegangen, die man am Ende erweitert.

Grundsätzlich bezeichnet man in Pascal mit einer Liste eine lineare Datenstruktur, die durch Zeiger gebildet wird. Linear bedeutet in diesem Zusammenhang, daß jedes Element genau einen Vorgänger und Nachfolger besitzt.

Folgende Operationen sind in Listen möglich:

1. Start mit der leeren Liste
2. Erweitern der Liste
3. Löschen in der Liste

Es gibt zahlreiche verschiedene Listentypen, die sich durch die Art der Verzeigerung unterscheiden. Wenn Sie noch einmal Bild 15 betrachten, werden Sie feststellen, daß man mit der Operation

```
KUNDENEU:= KUNDENEU↑.NAECHSTER
```

ohne Probleme die Liste *vorwärts* durchlaufen kann. Andererseits ist es (ohne einen Zugriff auf andere Zeiger) nicht möglich, von LETZTERKUNDE zurück zum Vorgänger in der Liste zu gelangen. So bestimmt also die Zeigerstruktur die Art der möglichen Zugriffe auf eine Liste.

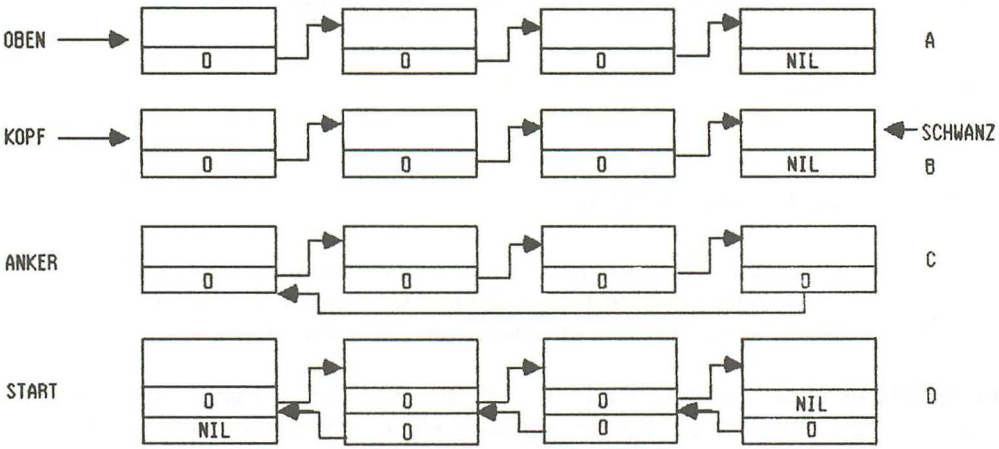


Bild 16: *Listenstrukturen*

In Bild 16 sind die wichtigsten Listentypen grafisch dargestellt.

A Kellerspeicher (stack, Stapelspeicher)
B Schlange (queue)
C Ringspeicher
D Doppelt verkettete Liste

Bei einem Kellerspeicher fügt man Elemente bei OBEN ein und löscht sie auch dort wieder. Weil dadurch das zuletzt eingefügte Element zuerst gelöscht wird, heißt ein Kellerspeicher auch LIFO-Speicher (last-in-first-out).

Bei einer Schlange fügt man Elemente bei SCHWANZ ein und löscht sie bei KOPF. Schlangen heißen auch FIFO-Speicher (first-in-first-out).

In einigen Anwendungen sind Ringspeicher sinnvoll. Hierbei ist keine Ordnung auf den Elementen definiert. Jedes Element ist Nachfolger eines anderen. ANKER wird nur benötigt, um einen Zugriff auf ein Element des Ringes zu besitzen. Wäre ANKER nicht vorhanden, so könnte man nämlich **keines** der Elemente über einen Bezeichner (z.B. mit ANKER↑) erreichen!

Relativ aufwendige Operationen erfordert die Konstruktion einer doppelt verketteten Liste. Ein wesentlicher Vorteil ist die Tatsache, daß man sich in beiden Richtungen in der Liste bewegen kann.

Natürlich können wir nicht alle Typen in diesem Buch behandeln. Dieser Überblick sollte Ihnen nur die grundsätzlichen Probleme beim Aufbau von dynamischen Strukturen zeigen. Die Bearbeitung von Listen besteht also größtenteils im Verfolgen von Zeigerketten.

In Listing 33 ist ein komplettes Programm angegeben, das die am Anfang des Abschnitts erwähnte Kundenliste implementiert. Alle Funktionen sind zu Modulen zusammengefaßt und ausführlich kommentiert, so daß Sie die einzelnen Operationen nachvollziehen können. Auf einige Details sollten Sie achten:

Will man in einer Liste ein Element löschen oder einfügen, so muß man das Feld NAECHSTER beim **Vorgänger** korrigieren. Deshalb werden in der Suchroutine VORHANDEN zwei Zeiger verwendet. Dabei hinkt der Zeiger Z beim Durchlaufen der Liste immer ein Record hinter dem Zeiger Z1 her.

Normalerweise muß man die erste Einfügung in der Liste und das Löschen des letzten Elementes in der Kette explizit programmieren, da hierbei andere Zeiger umgesetzt werden müssen als bei allen anderen Operationen. Um diese Sonderbehandlungen zu vermeiden, wird im Programm die Liste um ein unbenutztes erstes und letztes Element erweitert.

Dieses letzte Element wird auch zur Aufnahme einer Marke bei der Suchroutine VORHANDEN verwendet (siehe auch Abschnitt 2.9 über die Suche im Array).

Interessant ist vielleicht noch die folgende Variablenangabe in der Prozedur LOESCHEN:

```
VOR↑.NAECHSTER:= VOR↑.NAECHSTER↑.NAECHSTER
```

Auf der rechten Seite des Zuweisungsoperators wird zweimal dereferenziert: Das Ergebnis ist also der Zeiger, der im Feld NAECHSTER des Nachfolgers von VOR↑ steht.

```
PROGRAM KUNDENLISTE (INPUT, OUTPUT);
(* BEISPIEL FUER DIE VERWALTUNG EINER LISTE MIT ZEIGERN.  *)
(* DIE DATEN WERDEN STAENDIG SORTIERT IN EINER LISTE GE-  *)
(* HALTEN. JEDER RECORD BESITZT DAZU EINEN ZEIGER AUF     *)
(* DEN ALPHABETISCHEN NACHFOLGER. UM DAS EINFUEGEN UND    *)
(* LOESCHEN EINFACH ZU GESTALTEN, BESITZT DIE LISTE JE    *)
(* EIN LEERES ELEMENT AM ANFANG UND ENDE.                 *)

  CONST LEN = 10;                (* LAENGE EINES NAMENS    *)

  TYPE STRING = ARRAY [1..LEN] OF CHAR;
       KUNDENZEIGER = ↑ KUNDE;
       KUNDE  = RECORD
                  NAME     : STRING;
                  KNUMMER  : INTEGER;
                  NAECHSTER: KUNDENZEIGER;
                END;

  VAR KOPF: KUNDENZEIGER;        (* KOPF DER KUNDENLISTE   *)
      ENDE: KUNDENZEIGER;        (* ENDE DER KUNDENLISTE   *)
      CH  : CHAR;                (* BENUTZEREINGABE        *)

  PROCEDURE READSTRING(VAR S: STRING);
  (* STRING MIT LEN ZEICHEN VON DER TASTATUR LESEN.       *)
    VAR I: INTEGER;
        C: CHAR;
  BEGIN
    REPEAT READ(C) UNTIL C<>" ";(* VORLAUFENE LEERZEICHEN *)
    I:= 1;                     (* IGNORIEREN             *)
    REPEAT                     (* LEN ZEICHEN ODER BIS   *)
      S[I]:= C; I:= I+1;       (* ZUM ZEILENENDE LESEN   *)
      READ(C)
    UNTIL (I>LEN) OR EOLN;
    WHILE I<=LEN DO            (* S MIT LEERZEICHEN AUF- *)
      BEGIN                    (* FUELLEN                *)
        S[I]:= " "; I:=I+1
      END;
    WRITELN
  END; (* READSTRING *)

  FUNCTION VORHANDEN(S:STRING; VAR Z:KUNDENZEIGER):BOOLEAN;
  (* SUCHT NAME (S) IN DER LISTE. ERGEBNIS=TRUE, FALLS    *)
  (* S GEFUNDEN WURDE. Z ZEIGT BEI RUECKKEHR IMMER AUF    *)
  (* DIE POSITION DES ALPHABETISCHEN VORGAENGERS.         *)
    VAR Z1: KUNDENZEIGER;     (* Z1 STEHT IMMER EIN RECORD*)
                              (* WEITER ALS DER ZEIGER Z  *)
  BEGIN
    Z:= KOPF; Z1:= KOPF↑.NAECHSTER;
    ENDE↑.NAME:= S;           (* MARKE AM LISTENENDE      *)
    WHILE Z1↑.NAME<S DO
      BEGIN
        Z:= Z1; Z1:= Z1↑.NAECHSTER
      END;
    VORHANDEN:= (Z1↑.NAME=S) AND (Z1<>ENDE)
  END; (* VORHANDEN *)
```

```
PROCEDURE DRUCKE(Z: KUNDENZEIGER);
(* DRUCKE DEN INHALT DES REFERENZIERTEN RECORDS          *)
BEGIN
 WITH Z↑ DO
   WRITELN("NAME:",NAME:LEN+2,"  NUMMER:",KNUMMER:5)
END; (* DRUCKE *)

PROCEDURE EINGABE;
(* EINGABE EINES NEUEN KUNDENRECORDS                     *)
  VAR N  : STRING;
      NEU: KUNDENZEIGER;     (* ZEIGER AUF NEUEN RECORD  *)
      VOR: KUNDENZEIGER;     (* ZEIGER AUF ALPABETISCHEN *)
                             (* VORGAENGER IN DER LISTE  *)
BEGIN
  WRITE("NAME:"); READSTRING(N);
  IF VORHANDEN(N,VOR) THEN
     WRITELN(N," IST BEREITS KUNDE!")
  ELSE
     BEGIN
      NEW(NEU);              (* NEUEN RECORD BESORGEN    *)
      WRITE("KUNDENNUMMER:");
      READLN(NEU↑.KNUMMER); (* UND BELEGEN              *)
      NEU↑.NAME:= N;
      NEU↑.NAECHSTER:= VOR↑.NAECHSTER;
      VOR↑.NAECHSTER:= NEU; (* NEU NACH VOR EINFUEGEN   *)
    END
END; (* EINGABE *)

PROCEDURE AUSGABE;
(* AUSGABE EINES KUNDENRECORDS                           *)
  VAR N  : STRING;
      VOR: KUNDENZEIGER;     (* ZEIGER AUF ALPHABETISCHEN *)
                             (* VORGAENGER IN DER LISTE  *)
BEGIN
  WRITE("NAME:"); READSTRING(N);
  IF VORHANDEN(N,VOR) THEN
     DRUCKE(VOR↑.NAECHSTER)
  ELSE
     WRITELN(N,"NICHT ALS KUNDE GESPEICHERT!")
END; (* AUSGABE *)

PROCEDURE LOESCHEN;
  VAR N  : STRING;
      VOR: KUNDENZEIGER;     (* VORGAENGER IN DER LISTE  *)
BEGIN
  WRITE("NAME:"); READSTRING(N);
  IF VORHANDEN(N,VOR) THEN
     BEGIN
      WRITELN("GELOESCHT WURDE:");
      DRUCKE(VOR↑.NAECHSTER);
                             (* NACHFOLGER VON VOR AUS   *)
                             (* DER LISTE STREICHEN:     *)
      VOR↑.NAECHSTER:= VOR↑.NAECHSTER↑.NAECHSTER;
     END
  ELSE
     WRITELN(N,"NICHT ALS KUNDE GESPEICHERT!")
END; (* LOESCHEN *)
```

```
  PROCEDURE TABELLE;
  (* DRUCKE EINE ALPHABETISCHE LISTE ALLER KUNDEN          *)
    VAR Z:KUNDENZEIGER;
  BEGIN
    Z:= KOPF↑.NAECHSTER;        (* Z AUF ANFANG DER LISTE    *)
    WHILE Z<>ENDE DO            (* SOLANGE NICHT LETZTEN     *)
      BEGIN                     (* (LEEREN) RECORD ERREICHT:*)
        DRUCKE(Z);
        Z:=Z↑.NAECHSTER         (* ZUM NAECHSTEN KUNDEN      *)
      END;
  END; (* TABELLE *)

BEGIN (* HAUPTPROGRAMM *)
  NEW(KOPF); NEW(ENDE);         (* ANFANG UND ENDE BILDEN    *)
  KOPF↑.NAECHSTER:=ENDE;        (* LISTE IST LEER            *)
  REPEAT                        (* EINGABESCHLEIFE           *)
    WRITELN("E INGABE");
    WRITELN("A USGABE");
    WRITELN("L OESCHEN");
    WRITELN("T ABELLE");
    WRITELN("X BEENDEN");
    READLN(CH);
    CASE CH OF
      "T": TABELLE;
      "E": EINGABE;
      "A": AUSGABE;
      "L": LOESCHEN;
      "X": ;
     ELSE WRITELN("UNGUELTIGE WAHL")
    END;
  UNTIL CH="X";
END.
```

Listing 33: *Programm Kundenliste*

Zum Abschluß des Abschnitts sollen Sie noch ein Standardverfahren kennenlernen, mit dem man den Speicherplatz, der durch das Löschen von Records frei wird, wiederverwenden kann. Die Idee besteht darin, die (logisch) gelöschten Records zu einer neuen Liste, der **Freispeicherliste**, zu verketten. Vor jedem Aufruf der Prozedur NEW prüft man dann, ob sich nicht ein unbenutzter Record in der Freispeicherliste befindet.

Einfügungen und Löschungen in der Freispeicherliste erfolgen am einfachsten am selben Ende, so daß diese Liste also ein LIFO (stack, Stapel) ist. Um diese Freispeicherverwaltung in das Programm Kundenliste zu integrieren, muß man folgende Änderungen vornehmen: Zunächst deklariert man einen Zeiger auf den Kopf der Freiliste.

```
FREI: KUNDENZEIGER;
```

Anschließend werden die eigentlichen Prozeduren zur Verwaltung der Freiliste definiert:

```
PROCEDURE NEWKUNDE(VAR Z:KUNDENZEIGER);
(* Liefere Zeiger auf neuen Kundenrecord*)
BEGIN
  IF FREI = NIL THEN
    (* Freispeicher ist leer: *)
    NEW(Z)
  ELSE
    BEGIN
      (* Entferne ersten Record aus Freispeicher*)
      Z:= FREI; FREI:= FREI↑.NAECHSTER
    END
END; (* NEWKUNDE *)

PROCEDURE DISPOSEKUNDE(Z: KUNDENZEIGER);
(* Speicherplatz von Z ist freigeworden,*)
(* Erweitere die Freispeicherliste      *)
BEGIN
  Z↑.NAECHSTER:= FREI;
  FREI:= Z
END; (* DISPOSEKUNDE *)
```

Jetzt müssen die Routinen nur korrekt aufgerufen werden. Dazu ersetzt man den Aufruf NEW(NEU) durch NEWKUNDE(NEU). Um in der Prozedur LOESCHEN den Nachfolger von VOR zu löschen, merkt man sich zunächst in einer Variablen ALT den Zeiger auf das zu löschende Objekt. Dann kann man den Record aus der Verzeigerung der Kundenliste entfernen und zum Schluß mit DISPOSEKUNDE(ALT) den Record ALT↑ in die Freispeicherliste einfügen:

```
PROCEDURE LOESCHEN;
VAR ...
    ALT: KUNDENZEIGER;
...
  ALT:= VOR↑.NAECHSTER;
  VOR↑.NAECHSTER:= VOR↑.NAECHSTER↑.NAECHSTER;
  DISPOSEKUNDE(ALT)
...
```

Natürlich muß die Freispeicherliste am Programmanfang korrekt initialisiert werden. Da sie zu diesem Zeitpunkt noch leer ist, erhält der Zeiger auf den Listenanfang den Wert NIL:

```
FREI:= NIL
```

Dieses Verfahren der Verwaltung von freigewordenem Speicher ist sehr effizient, so daß man meist die Verwendung systemspezifischer Speicherverwaltungsprozeduren (DISPOSE, MARK, RELEASE) vermeiden kann.

Die Prozeduren MARK und RELEASE in Pascal 1.4 sind in der Dokumentation in Kapitel 4 beschrieben.

2.18.2 Bäume

Zum Abschluß dieser Einführung in die Programmiersprache Pascal soll noch ein Beispiel für eine nichtlineare dynamische Datenstruktur mit Zeigern gegeben werden: Ein **Baum** ist (in der Graphentheorie) ein Graph mit einem Eingang, in dem jeder Knoten auf genau einem Weg vom Eingang erreicht werden kann (siehe Bild 17).

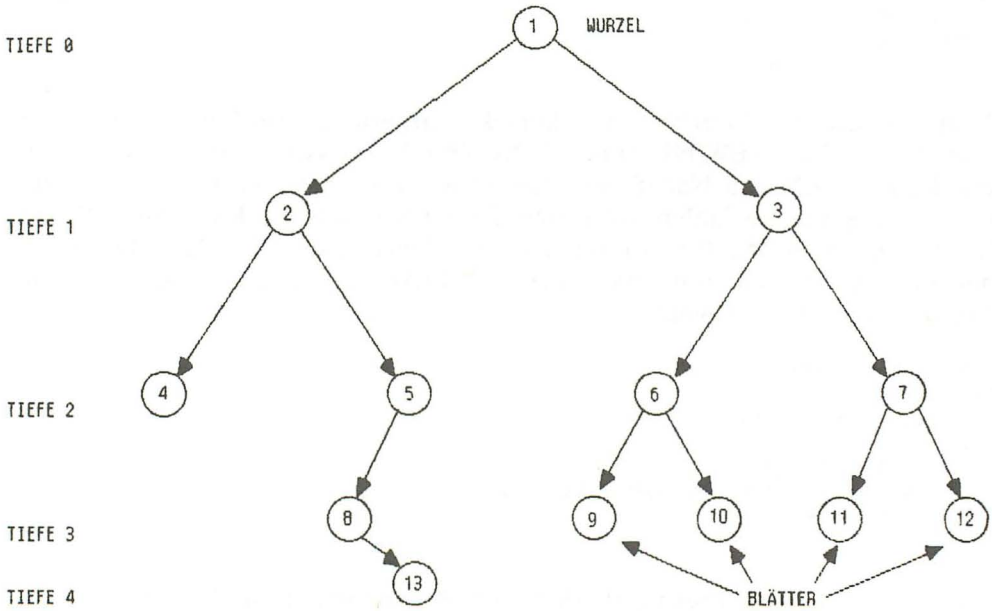


Bild 17: *Ein Baum*

Bäume zeichnet man üblicherweise mit der **Wurzel** (dem Eingang) nach oben. Jeder Knoten besitzt eine gewisse **Tiefe**, das ist die Distanz zum Eingang (hier also Knoten 1). Knoten ohne Nachfolger bezeichnet man als

Blätter. Jeder innere Knoten hat eine gewisse Anzahl an direkten Nachfolgern, die selbst Wurzeln von **Teilbäumen** sind. So besitzt der Wurzelknoten (1) zwei direkte Nachfolger (2 und 3), wobei z.B. 3 die Wurzel des Teilbaumes aus den Knoten 3, 6, 7, 9, 10, 11 und 12 bildet. Die Maximalanzahl der direkten Nachfolger, die ein Knoten in einem Baum besitzt, heißt der **Grad** des Baumes.

Wir wollen uns nur mit binären Bäumen, also mit Bäumen des Grades 2 beschäftigen: In ihnen besitzt ein innerer Knoten 1 oder 2 Nachfolger, während ein Blatt 0 Nachfolger besitzt. Außerdem definieren wir eine **Ordnung** auf den Knoten des Baumes. Für jeden Knoten K im Baum gelten folgende Relationen:

1. Alle Knoten im linken Teilbaum mit der Wurzel K sind kleiner als K.
2. Alle Knoten im rechten Teilbaum mit der Wurzel K sind größer als K.

Mit dieser Regel ergibt sich in Bild 17 für die Wurzel folgende Relation:

2, 4, 5, 8, 13 < 1 < 3, 6, 7, 9, 10, 11, 12

Wendet man diese Regeln auch auf alle Knoten in den Teilbäumen an, so erhält man eine vollständige Ordnung.

4 < 2 < 8 < 13 < 5 < 1 < 9 < 6 < 10 < 3 < 11 < 7 < 12

Nun haben wir alle Begriffe beisammen, um unsere Kundenverwaltung in einem binären Baum zu organisieren. Wiederum sollen alle Kunden alphabetisch sortiert gespeichert werden, um ohne Nachsortieren eine nach Namen geordnete Liste auszugeben. Vor allen Dingen wird die Ordnung jedoch auch benutzt, um einen Kunden in kurzer Zeit zu finden, ohne (wie in einer Liste) alle Knoten zu untersuchen.

Die Knoten eines Baumes werden in Pascal durch einen Record dargestellt. Wir benutzen also wieder die Kundenrecords aus dem letzten Kapitel. Jedoch erhält jeder Kunde **zwei** Nachfolger. Die Felder L und R enthalten deshalb Zeiger auf den linken und rechten Nachfolger im Baum.

```
TYPE KUNDENZEIGER = ↑ KUNDE;
     KUNDE = RECORD
               NAME    : ARRAY [1..10] OF CHAR;
               KNUMMER: INTEGER;
               L,R     :KUNDENZEIGER;
             END;
VAR  WURZEL: KUNDENZEIGER;
```

Die Operationen mit dieser Struktur sind in Listing 34 beschrieben. Am Programmanfang ist der Baum leer:

```
WURZEL:= NIL
```

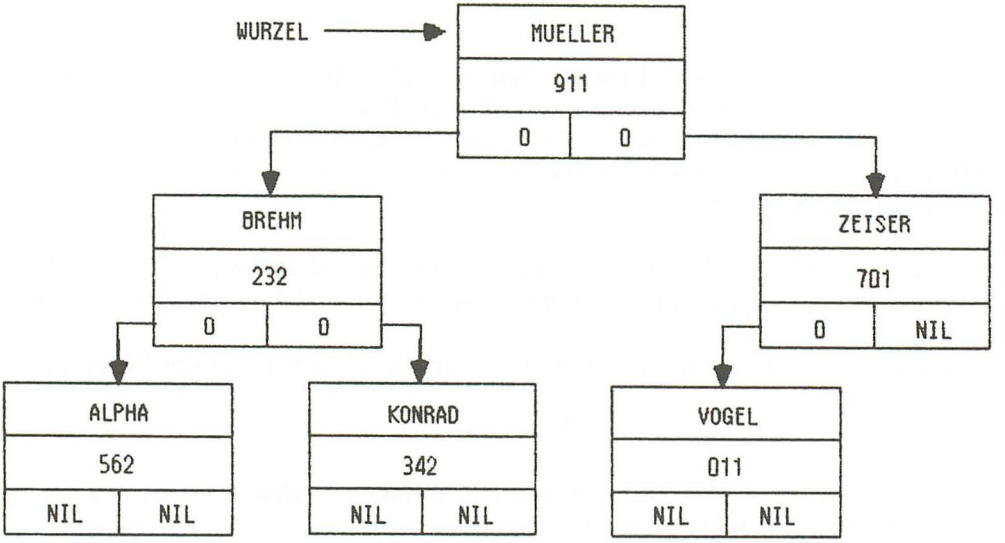


Bild 18: *Baum mit Kundenrecords*

Beim Einfügen müssen wir die alphabetische Reihenfolge im Baum beachten. Hierzu wandern wir ausgehend von der Wurzel zu den Blättern. Dabei vergleichen wir in jeder Tiefe den Namen des neuen Kunden mit dem Namen in den Knoten des Baumes. Ist der Name *größer*, so müssen wir die Einfügung im rechten Teilbaum ausführen, sonst verfolgen wir den Zeiger zum linken Teilbaum.

Um *Kunze* einzufügen, bestimmen wir so den folgenden Weg:

```
Kunze < Mueller      gehe links
Kunze > Brehm        gehe rechts
Kunze > Konrad       gehe rechts
```

Da kein rechter Nachfolger von Konrad existiert (der Zeiger R ist NIL), können wir jetzt Kunze als rechten Nachfolger von Konrad einfügen.

Diese Strategie beschreibt die rekursive Prozedur EINFUEGEN. Sie wird mit zwei Parametern aufgerufen: NEU ist ein Record vom Typ Kunde, um

den der Baum erweitert werden soll. Der Variablenparameter Z ist ein Zeiger auf die Wurzel des Teilbaumes, in den NEU eingefügt werden soll. Bitte beachten Sie, daß hier ein Variablenparameter erforderlich ist, um bei einer Einfügung (Z=NIL) den Zeiger auf das neue Element in der aufrufenden Umgebung zu ändern.

Bei der Suche fällt auf, daß man in jedem Schritt die Größe des noch zu untersuchenden Teilbaumes halbiert. Somit hat man im Idealfall in einem Baum mit N Knoten nach log(N) Schritten die Einfügeposition bestimmt. Natürlich muß man dafür sorgen, daß der Baum **ausgeglichen** bleibt. Würde man nämlich ständig Einfügungen am rechten Blatt vornehmen, so hätte der entstehende Baum die Form einer Liste, so daß für N Knoten die Tiefe N statt log(N) beträgt.

Zur Ausgabe der alphabetisch geordneten Tabelle muß man den Baum in der vorgegebenen Ordnung durchlaufen. Dies geschieht am elegantesten mit der rekursiven Prozedur INORDER. Der Parameter Z gibt die Wurzel des Teilbaumes an, der gedruckt werden soll. Die oben beschriebene Ordnung verlangt eine Ausgabe in der folgenden Reihenfolge:

1. Ausgabe aller Knoten im linken Teilbaum (Z↑.L).
2. Ausgabe des Wurzelknotens Z↑.
3. Ausgabe aller Knoten im rechten Teilbaum (Z↑.R).

Da die Wurzel in der Ordnung zwischen dem linken und rechten Teilbaum liegt, heißt diese Ordnung inorder (im Gegensatz zu preorder und postorder).

Löschungen eines inneren Knotens im geordneten binären Baum erfordern etwas genauere Überlegungen, damit die Ordnung zwischen den übriggebliebenen Knoten erhalten bleibt. Ohne nähere Erläuterungen ist im Listing 34 ein Löschalgorithmus angegeben.

Sollten Sie Interesse an solchen Datenstrukturen und Algorithmen gefunden haben, können Sie sich, ausgerüstet mit den Kenntnissen aus diesem Buch, der Fachliteratur über Systematische Programmierung (Bücher 2, 5, 6, 7, 8 Anhang E) zuwenden.

```
PROGRAM KUNDENBAUM (INPUT, OUTPUT);
(* VERWALTUNG DER KUNDENDATEN IN EINEM NACH NAMEN          *)
(* GEORDNETEN BINAERBAUM.                                   *)

  CONST LEN = 10;                (* LAENGE EINES NAMENS    *)
```

```
TYPE STRING = ARRAY [1..LEN] OF CHAR;
     KUNDENZEIGER = ↑ KUNDE;
     KUNDE  = RECORD
                NAME     : STRING;
                KNUMMER  : INTEGER;
                L, R     : KUNDENZEIGER;
              END;

VAR WURZEL: KUNDENZEIGER;      (* WURZEL DES BAUMES       *)
    CH    : CHAR;              (* FUNKTIONS-AUSWAHL       *)

PROCEDURE READSTRING(VAR S: STRING);
(* STRING MIT LEN ZEICHEN VON DER TASTATUR LESEN.        *)
  VAR I: INTEGER;
      C: CHAR;
BEGIN
  REPEAT READ(C) UNTIL C<>" ";(* VORLAUFENE LEERZEICHEN *)
  I:= 1;                      (* IGNORIEREN             *)
  REPEAT                      (* LEN ZEICHEN ODER BIS   *)
    S[I]:= C; I:= I+1;        (* ZUM ZEILENENDE LESEN   *)
    READ(C)
  UNTIL (I>LEN) OR EOLN;
  WHILE I<=LEN DO             (* S MIT LEERZEICHEN AUF- *)
    BEGIN                     (* FUELLEN                *)
      S[I]:= " "; I:=I+1
    END;
  WRITELN
END; (* READSTRING *)

PROCEDURE DRUCKE(Z: KUNDENZEIGER);
(* DRUCKE DEN INHALT DES REFERENZIERTEN RECORDS          *)
BEGIN
 WITH Z↑ DO
   WRITELN("NAME:",NAME:LEN+2,"  NUMMER:",KNUMMER:5)
END; (* DRUCKE *)

PROCEDURE EINGABE;
(* EINGABE EINES NEUEN KUNDENRECORDS                     *)
  VAR K: KUNDE;

  PROCEDURE EINFUEGEN(NEU:KUNDE; VAR Z:KUNDENZEIGER);
  (* FUEGE NEU AN DER KORREKTEN POSITION IM TEILBAUM MIT*)
  (* DER WURZEL Z EIN.                                  *)
  BEGIN
    IF Z=NIL THEN           (* TEILBAUM IST LEER:      *)
      BEGIN                 (* FUEGE NEU ALS BLATT EIN *)
        NEW(Z);Z↑:= NEU;    (* BELEGE Z↑ MIT NAME  UND *)
        Z↑.L:= NIL;         (* KUNDENNUMMER. Z↑ HAT   *)
        Z↑.R:= NIL          (* KEINE NACHFOLGER !      *)
      END
    ELSE
      BEGIN                 (* VERGLEICH DER SCHLUESSEL:*)
        IF NEU.NAME=Z↑.NAME THEN
          WRITELN(NEU.NAME," IST BEREITS KUNDE!")
        ELSE
          IF NEU.NAME<Z↑.NAME THEN
            EINFUEGEN(NEU,Z↑.L) (* IM LINKEN  ODER      *)
```

```
          ELSE
            EINFUEGEN(NEU,Z↑.R) (* IM RECHTEN TEILBAUM  *)
      END                       (* EINFUEGEN           *)
  END; (* EINFUEGEN *)
BEGIN  (* EINGABE *)
  WRITE("NAME:"); READSTRING(K.NAME);
  WRITE("KUNDENNUMMER:"); READLN(K.KNUMMER);
  EINFUEGEN(K, WURZEL);     (* K IM BAUM EINFUEGEN      *)
END; (* EINGABE *)

PROCEDURE INORDER(Z: KUNDENZEIGER);
(* DRUCKE IN ALPHABETISCHER REIHENFOLGE DIE KNOTEN DES  *)
(* TEILBAUMES MIT WURZEL Z.                             *)
BEGIN
  IF Z<>NIL THEN            (* TEILBAUM IST NICHT LEER: *)
    BEGIN
      INORDER(Z↑.L);        (* DRUCKE LINKEN TEILBAUM   *)
      DRUCKE (Z);           (* DIE WURZEL SELBST UND    *)
      INORDER(Z↑.R)         (* DANN DEN RECHTEN TEILBAUM*)
    END
END; (* TABELLE *)

PROCEDURE LOESCHE;
  VAR NAME: STRING;

  PROCEDURE ENTFERNE(N: STRING; VAR Z: KUNDENZEIGER);
  (* ENTFERNE DEN KUNDEN MIT NAME N AUS DEM TEILBAUM    *)
  (* MIT DER WURZEL Z                                   *)

    PROCEDURE HOLEHOCH(VAR Z1: KUNDENZEIGER);
    (* ERSETZE Z DURCH DEN GROESSTEN WERT IM LINKEN     *)
    (* TEILBAUM Z↑L                                     *)
    BEGIN
      IF Z1↑.R=NIL THEN
        BEGIN               (* KOPIERE FELDER NACH Z↑   *)
          Z↑.NAME   := Z1↑.NAME;
          Z↑.KNUMMER:= Z1↑.KNUMMER;
          Z1:=Z1↑.L         (* ERSETZE Z1 DURCH SEINEN  *)
                            (* LINKEN NACHFOLGER        *)
        END
      ELSE                  (* RECHTS WEITERSUCHEN:     *)
        HOLEHOCH(Z1↑.R)
    END; (* HOLEHOCH *)

  BEGIN  (* ENTFERNE *)
    IF Z=NIL THEN
      WRITELN(N," IST NICHT GESPEICHERT!")
    ELSE
      IF N=Z↑.NAME THEN     (* ERSETZE Z DURCH EINEN    *)
        BEGIN               (* SEINER NACHFOLGER        *)
          IF Z↑.L= NIL THEN Z:=Z↑.R ELSE
          IF Z↑.R= NIL THEN Z:=Z↑.L
          ELSE  HOLEHOCH(Z↑.L)
        END
      ELSE                  (* SUCHE IN DEN TEILBAEUMEN *)
        IF N<Z↑.NAME THEN ENTFERNE(N,Z↑.L)
                     ELSE ENTFERNE(N,Z↑.R)
```

```
    END;(* ENTFERNE *)
  BEGIN (* LOESCHE *)
    WRITE("NAME:"); READSTRING(NAME);
    ENTFERNE(NAME,WURZEL)      (* LOESCHE KUNDEN IM BAUM   *)
  END; (* LOESCHE *)

BEGIN (* HAUPTPROGRAMM *)
  WURZEL:= NIL;                (* BAUM IST LEER            *)
  REPEAT                       (* EINGABESCHLEIFE          *)
    WRITELN("T ABELLE");
    WRITELN("E RWEITERN");
    WRITELN("L OESCHEN");
    WRITELN("X BEENDEN");
    READLN(CH);
    CASE CH OF
      "T": INORDER(WURZEL);    (* DRUCKE GESAMTEN BAUM     *)
      "E": EINGABE;
      "L": LOESCHE;
      "X": ;
     ELSE WRITELN("UNGUELTIGE WAHL")
    END;
  UNTIL CH="X";
END.
```

Listing 34: *Kundenverwaltung in einem Baum*

Aufgaben

1. Versuchen Sie, eine Kundenliste ohne leere Records am Anfang und Ende der Liste zu verwalten. Sollten Sie nicht mehr weiterkommen, haben Sie zumindest den Sinn dieser Hilfsrecords erkannt.

2. Um zu prüfen, ob Sie die Operationen im Listing 34 im großen und ganzen verstanden haben, sollten Sie das Programm so ändern, daß die Records nach Kundennummer sortiert im Baum gespeichert werden.

3. Schreiben Sie eine rekursive Prozedur SWAP, die in einem binären Baum den linken und rechten Nachfolger jedes inneren Knotens vertauscht.

4. Schreiben Sie eine rekursive Prozedur REVERSE, die eine Liste invertiert, so daß der letzte Record als erster in der neuen Liste erscheint. (Diese rekursive Lösung ist nur für kurze Listen geeignet.)

3 Tips und Tricks

3.1 Nützliche Pascal-Routinen

Im Gegensatz zu Kapitel 2 stehen in diesem Kapitel die Besonderheiten des C 64 im Vordergrund. Dabei sollen nicht alle POKE-, PEEK- und SYS-Befehle, die seit Jahren verschiedene Zeitschriften zum C 64 füllen, in Pascal formuliert werden, sondern nur exemplarisch der Zugriff auf das Betriebssystem und die Floppy gezeigt werden.

Files

Sollten Sie in BASIC bereits mit Dateien gearbeitet haben, werden Sie sicher keine Probleme mit den OPEN- und CLOSE-Befehlen in Pascal haben. Haben Sie jedoch erst durch Abschnitt 2.16 Interesse an Files gefunden, sind sicherlich die folgenden Hinweise angebracht:

Die Floppy besitzt ein eigenes Betriebssystem, das die Files auf der Diskette verwaltet. Um ein File zum Lesen oder Schreiben zu eröffnen, muß man den Filenamen auf der Diskette angeben. Außerdem nennt man eine Sekundäradresse, die im Bereich von 0 bis 15 liegt. Um auf ein geöffnetes File Bezug zu nehmen, verwendet man diese Sekundäradresse.

Dabei besitzen die Sekundäradressen 0, 1 und 15 eine besondere Bedeutung und sollten nicht für sequentielle Dateien verwendet werden. Des weiteren müssen gleichzeitig eröffnete Dateien verschiedene Sekundäradressen er-

halten. Beim OPEN-Befehl geben Sie außerdem an, ob von der Datei gelesen werden soll, ob die Datei neu angelegt werden soll oder ob eine bestehende Datei am Ende erweitert werden soll:

```
OPEN(F,8,3,"EINGABE,SEQ,READ")        entspricht RESET(F)
OPEN(G,8,4,"AUSGABE,SEQ,WRITE")       entspricht REWRITE(G)
OPEN(H,8,5,"PROTOKOLL,SEQ,APPEND")    sequentiell erweitern
```

Der dritte OPEN-Befehl zeigt, wie man ein bestehendes File um Komponenten am Ende erweitern kann. Diese Operation läßt sich prinzipiell nicht mit den Standardbefehlen RESET und REWRITE realisieren.

Existiert bei der Ausführung des zweiten OPEN-Befehls bereits eine Datei mit dem Namen AUSGABE, so würde eine Fehlermeldung durch die Floppy erzeugt. Deshalb sollte man die alte Version zuvor löschen. Dabei gibt es zwei verschiedene Möglichkeiten: Einerseits kann man den Filenamen durch Voranstellen des *Klammeraffen*-Zeichens CHR(64) und eines Doppelpunktes erweitern. Dann wird am Ende der Ausgabe auf das File AUSGABE die alte Version aus dem Inhaltsverzeichnis der Diskette gelöscht. Jedoch wird der Platz, den die alte Version belegte, nicht freigegeben. Deshalb kann beim Schreiben nicht die gesamte Speicherkapazität der Diskette ausgenutzt werden. Man löscht daher besser die alte Version des Files **vor** dem Eröffnen einer neuen Datei. Dafür kann man den Kommandokanal der Floppy (mit der Sekundäradresse 15) benutzen:

```
VAR KOMMANDO: TEXT;
...
OPEN(KOMMANDO,8,15);
WRITELN(KOMMANDO,"S0:AUSGABE");
OPEN(G,8,4,"AUSGABE,S,W")
```

Der Kommandokanal stellt die Schnittstelle des Programmes zur Floppy dar. Über ihn sendet man Befehle an das Betriebssystem der Floppy und empfängt Meldungen über eventuell aufgetretene Fehler. Ein Beispiel für die Abfrage des Kommandokanals ist die Prozedur DSTATUS im Programm RELATIV.P auf der Systemdiskette.

Jetzt kennen Sie den Zusammenhang zwischen Files in Pascal und den sequentiellen Dateien der Floppy. Mit diesem Wissen können Sie die Beispiele aus dem Floppy-Handbuch als Vorbild für eigene Programme in Pascal benutzen.

Für erfahrene Programmierer ist in Listing 35 ein Programm abgedruckt, das den Zugriff auf das Directory der Diskette zeigt. Es wird eine Datei der Floppy gelesen, die man normalerweise in BASIC mit LOAD"$",8 lädt.

Alle Filenamen werden mit Typ und Längenangabe in einer Tabelle gespeichert. Nach einer alphabetischen Sortierung können dann alle Namen formatiert ausgedruckt werden.

```
PROGRAM DISKSORT(INPUT,OUTPUT);
(*FORMATIERTER AUSDRUCK DES DISKINHALTES        *)
CONST MAXP=250;   (* LAENGE DER NAMENSTABELLE   *)
TYPE  FILETYP=(PRG,SEQ,USR,REL);
      EINTRAG=RECORD
               NAME: ARRAY[0..15]OF CHAR;
               TYP :FILETYP;
               BLK :0..999;
               ID  :ARRAY[0..1]OF CHAR;
              END;
VAR   P: INTEGER; WEITER: BOOLEAN;
      T:ARRAY[0..MAXP] OF EINTRAG;

 PROCEDURE READALL;
 (* DISKETTEN-DIRECTORY KOMPLETT EINLESEN        *)
 (* INFORMATIONEN IN T[P] ABLEGEN                *)
  VAR A,ID1,ID2: CHAR;
      I,N       : INTEGER;
      INF       : TEXT;
      ENDE      : BOOLEAN;

   FUNCTION BYTE:INTEGER;
   (* ZEICHEN LESEN UND IN BYTE WANDELN          *)
     VAR C:CHAR;
   BEGIN
     READ(INF,C);
     IF EOLN(INF) THEN BYTE:=13
                  ELSE BYTE:=ORD(C)
   END; (* BYTE *)

   FUNCTION NUMBER:INTEGER;
   (* L UND H-BYTE LESEN UND UMWANDELN           *)
   BEGIN
     NUMBER:= BYTE+256*BYTE
   END; (* NUMBER *)

 BEGIN(*READALL*)
  OPEN(INF,8,0,"$0");   (* INHALTSVERZEICHNIS   *)
  WRITELN;WRITELN;
  FOR I:=1 TO 32 DO     (* TITELZEILE AUSWERTEN:*)
   BEGIN READ(INF,A);
     IF I IN [9..24] THEN WRITE(A);
     IF I=27 THEN ID1:=A;
     IF I=28 THEN ID2:=A
   END;
  WRITELN; ENDE:= FALSE;
  WHILE (P<=MAXP) AND NOT ENDE DO
  BEGIN
   N:=NUMBER; N:= NUMBER;    (* N=ANZAHL BLOECKE*)
   (* ANFUEHRUNGSZEICHEN ODER"BLOCKS FREE"LESEN *)
   REPEAT
    ENDE:= EOF(INF)
```

```
  UNTIL (BYTE=34) OR ENDE;
  IF NOT ENDE THEN        (* ALLES EINTRAGEN:  *)
   WITH T[P] DO
    BEGIN I:=0;           (* EINTRAG FILENAME: *)
     REPEAT READ(INF,A); NAME[I]:=A; I:=I+1
     UNTIL A=CHR(34);
     I:=I-1;WHILE I<16 DO
      BEGIN NAME[I]:=" "; I:=I+1 END;
     REPEAT READ(INF,A);  (*EINTRAG TYP:       *)
     UNTIL A IN ["P","S","U","R"];
     CASE A OF
       "P":TYP:=PRG;
       "S":TYP:=SEQ;
       "U":TYP:=USR;
       "R":TYP:=REL
     END; (*CASE*)
     REPEAT UNTIL BYTE=0;    (*BIS ZEILENENDE  *)
     BLK:=N;
     ID[0]:=ID1; ID[1]:=ID2; (*EINTRAG DISK-ID:*)
     P:= P+1;
    END
 END;
 CLOSE(INF);
END; (* READALL *)

PROCEDURE QUICK(L,R:INTEGER);
(* TABELLE NACH NAMEN AUFSTEIGEND SORTIEREN    *)
 VAR I,J:INTEGER;
     X,W:EINTRAG;

BEGIN
 I:=L; J:=R; X:=T[(L+R)DIV 2];
 REPEAT
  WHILE T[I]<X DO I:=I+1;
  WHILE X<T[J] DO J:=J-1;
  IF I<=J THEN
  BEGIN
   W:=T[I];T[I]:=T[J];T[J]:=W;
   I:=I+1;J:=J-1
  END;
 UNTIL I>J;
 IF L<J THEN QUICK(L,J);
 IF I<R THEN QUICK(I,R)
END; (* QUICK *)

FUNCTION OK:BOOLEAN;
  VAR C:CHAR;
BEGIN
 WRITELN("  (JA ODER NEIN)"); WRITE("==>");
 REPEAT READLN(C) UNTIL C IN ["J","N"];
 OK:= C="J"
END;(* OK *)

PROCEDURE AUSGABE;
 VAR J, ZPROSEITE: INTEGR;
     PRT          : TEXT;
BEGIN
```

```
  WRITELN; WRITELN;
  WRITELN("DRUCKER BEREIT?");
  IF OK THEN
   BEGIN
    OPEN(PRT,4,0);
    WRITE("ZEILEN PRO SEITE: ");READLN(ZPROSEITE);
    FOR J:=0 TO P-1 DO
     WITH T[J] DO
      BEGIN
       IF J MOD (ZPROSEITE-2)=0 THEN
        BEGIN
         WRITE(PRT,"  NAME:                TYP ");
         WRITELN(PRT,"  BLK   ID  ");
         WRITE(PRT,"  #####                ### ");
         WRITELN(PRT,"  ###   ##  ");
        END;
       WRITE(PRT," ",NAME:17,"  ");
       CASE TYP OF
        PRG:WRITE(PRT,"PRG":4);
        SEQ:WRITE(PRT,"SEQ":4);
        USR:WRITE(PRT,"USR":4);
        REL:WRITE(PRT,"REL":4)
       END;
       WRITELN(PRT,"  ",BLK:4,"  ",ID:3,"  ");
      END;
     CLOSE(PRT)
    END
 END;(* AUSGABE *)

BEGIN(* MAIN *)
 P:=0;                          (* TABELLE LEER *)
 WRITELN(CHR(147),"DISK-SORT":24);
 WRITELN("          ":24);
 REPEAT
  WRITELN; WRITELN;
  WRITELN("WEITERE DISKETTEN?"); WEITER:=OK;
  IF WEITER THEN READALL;
 UNTIL NOT WEITER;
 WRITELN; WRITELN;
 WRITELN ("*** BITTE WARTEN ***");
 IF P>0 THEN QUICK(0,P-1);
 AUSGABE;
END. (* MAIN *)
```

Listing 35: *Directory lesen*

Es gibt noch einen weiteren Dateityp, der vom Betriebssystem der Floppy verwaltet wird. Es handelt sich dabei um **relative** Dateien. Sie können einerseits wie sequentielle Dateien Komponente für Komponente gelesen werden, andererseits besteht auch die Möglichkeit, gezielt auf einzelne Komponenten des Files zuzugreifen. Dies geschieht durch Angabe der Position der Komponente in der Datei: Alle Komponenten sind von 1 aufsteigend numeriert. Da in relativen Dateien alle Komponenten dieselbe Größe (in Bytes) besitzen, kann das Betriebssystem jede Komponente über

ihre **Record-Nummer** direkt adressieren. Die Verarbeitung relativer Dateien erfolgt also folgendermaßen:

1. Beim Eröffnen der Datei wird der Dateityp REL gewählt. Wird eine neue Datei angelegt, muß außerdem die Länge jeder Komponenten in Bytes angegeben werden.

2. Vor dem Schreiben einer Komponente kann man die Record-Nummer bestimmen, unter der die Daten gespeichert werden. Dies geschieht über den Kommandokanal der Floppy.

3. Vor dem Lesen einer Komponente kann man ebenfalls die Record-Nummer der Komponente angeben, die als nächste gelesen wird. Geschieht dies nicht, wird die Datei sequentiell gelesen.

Diese Operationen werden im Programm RELATIV.P auf der Systemdiskette demonstriert. Dabei ist die Positionierung auf die entsprechende Record-Nummer über dem Kommandokanal der Floppy als Prozedur definiert worden.

Bei der Arbeit mit relativen Dateien sind noch zwei Details zu beachten: Man muß beim OPEN-Befehl die Größe einer Komponente in Bytes angeben. Die Information über den Speicherplatz, den jeder Wert eines Typs in Pascal 1.4 benötigt, finden Sie in Abschnitt 4.4.2 unter dem Stichwort Datentypen. Im Beispielprogramm RELATIV.P ist diese Größe in der Konstanten RSIZE definiert. Schließlich erwartet das Betriebssystem der Floppy am Ende jeder Komponenten in einer relativen Datei das Zeichen CHR(13). Daher wird im Programm RELATIV.P jede Komponente um das Feld MARKE erweitert, in dem am Programmanfang das Zeichen CHR(13) gespeichert wird.

Systemadressen

In diesem Abschnitt werden einige Beispiele gegeben, die zeigen, wie man das Betriebssystem direkt manipuliert. Mit den Kenntnissen aus diesem Abschnitt können erfahrene Programmierer die zahlreichen BASIC-Programme, die mit PEEK und POKE Grafiken erzeugen, Sprites bewegen und die Tongeneratoren programmieren, in Pascal umschreiben. Anfänger können evtl. einige der Routinen als *black box* in eigenen Programmen benutzen.

Die erste Prozedur zeigt, wie man den freien Speicherbereich zwischen Heap und Stack bestimmt.

```
FUNCTION FREEMEM: INTEGER;
  VAR TOPOFSTACK: INTEGER[47];
  VAR HEAPPTR   : INTEGER[59];
BEGIN
 FREEMEM:= ADDU(HEAPPTR,-TOPOFSTACK)
END; (* FREEMEM *)
```

Um am Ende des Speichers N Byte zu reservieren, muß man vor dem ersten Aufruf der Standardprozedur NEW die folgende Prozedur GETMEM aufrufen. In diesem Speicherbereich kann man dann z.B. einen Bildschirmspeicher ablegen.

```
PROCEDURE GETMEM(N:INTEGER);
  VAR HEAPPTR[59];
BEGIN
 IF FREEMEM<N THEN
   BEGIN
     WRITELN("OUT OF MEMORY ERROR");
     HALT
   END
 ELSE HEAPPTR:=ADDU(HEAPPTR,-N);
END; (* GETMEM *)
```

In Abschnitt 4.3.3 wird außerdem beschrieben, wie man zwischen dem Laufzeitsystem und dem Objektprogramm einen Speicherbereich reserviert, in dem man z.B. Maschinenprogramme ablegen kann, die mit dem Pascal-Programm gespeichert werden sollen.

Die obigen Prozeduren und Funktionen benutzen die Möglichkeit, in Pascal 1.4 Variablen an Speicheradressen des Systems zu binden. Dies erlaubt auch die elegante Abfrage des Joysticks am Port 2: Jedes **nicht** gesetzte Bit in der Speicherstelle 65520 entspricht einer Bewegungsrichtung des Joysticks. Da in Pascal 1.4 Mengen als Bitvektoren dargestellt werden, kann man einfach auf einzelne Bits in einer Speicherstelle zugreifen.

```
PROGRAM TEST (INPUT,OUTPUT);
  TYPE JOY= SET OF (AUF,AB,LI,RE,FEUER);
  VAR JOYSTICK: JOY;

PROCEDURE UPDATEJOYSTICK;
  VAR JOYPORT:JOY[-9216]; (* =56320     *)
BEGIN
  (* negative Logik des Ports beachten:  *)
  JOYSTICK:=[AUF..FEUER]-JOYPORT;
END; (* UPDATEJOYSTICK *)

BEGIN
  REPEAT
    UPDATEJOYSTICK;
    IF RE  IN JOYSTICK THEN...;
    IF LI  IN JOYSTICK THEN...;
```

```
    IF AUF IN JOYSTICK THEN...;
    IF AB  IN JOYSTICK THEN...;
  UNTIL FEUER IN JOYSTICK;
END.
```

In einigen Anwendungen möchte man die Tastatur abfragen, ohne daß der Cursor am Bildschirm erscheint. Dazu kann man die Betriebssystem-Routine GETCH (get character) benutzen. Sie beginnt bei der Adresse $FFE4 =65508. Am Ende der Routine wird das Zeichen, das von der Tastatur eingelesen wurde, im Akkumulator des Mikroprozessors gespeichert. Zum Aufruf der Routine in Pascal benutzt man den SYS-Befehl mit der Integer-Zahl, die der Startadresse der Routine entspricht. In diesem Fall ist diese Adresse größer als MAXINT=32767, so daß man die Adresse im Zweierkomplement angeben muß. Dazu führt man folgende Berechnung durch:

```
$FFE4 = 65508 = -28 (65536 - 65508) im 2er-Komplement
```

Bei SYS wird der Akkumulator in der Speicherzelle 780 (dezimal) gespeichert (s. Abschnitt 4.4.4.7). Somit ergibt sich folgende Funktion:

```
FUNCTION GETKEY:CHAR;
BEGIN
  SYS(-28); (* $FFE4 *)
  GETKEY:= CHR(PEEK(780)); (* Zeichen im Akku *)
END;
[REPEAT]
UNTIL GETKEY<>CHR(0);
```

Die Funktion liefert als Ergebnis das Zeichen CHR(0), falls keine Taste gedrückt wurde, so daß die obige Repeat-Schleife durch die Betätigung einer beliebigen Taste beendet wird.

Um auf die vom Betriebssystem verwaltete Systemzeit zuzugreifen, kann man folgende Anweisungen verwenden. Zunächst wird der 24-Bit-Zähler zurückgestellt. Anschließend wird der Inhalt des Zählers byteweise ausgelesen.

```
PROGRAM TIME(INPUT,OUTPUT);
 CONST TI=160; (* Zähler in Z-Page *)

BEGIN
  POKE(TI  ,0);
  POKE(TI+1,0);
  POKE(TI+2,0);   (* Zähler := 0 *)
  REPEAT
   WRITELN(PEEK(TI),PEEK(TI+1),PEEK(TI+2))
  UNTIL PEEK(TI+1)=5;
END.
```

Als letztes Beispiel wollen wir noch die Benutzung von Routinen für reelle Zahlen vorstellen. Wir wollen den Wert einer reellen Variablen in eine Zeichenfolge S umwandeln.

Im Interpreter für BASIC ist ein Unterprogramm enthalten, das eine reelle Zahl in eine entsprechende Zeichenfolge umwandelt. Die reelle Zahl muß vor dem Aufruf in einem sogenannten Fließkomma-Akkumulator abgelegt werden. Dieser wird auch vom Pascal-Laufzeitsystem benutzt, wobei jede arithmetische Operation mit reellen Zahlen ihr Ergebnis dort ablegt. Deshalb führt die folgende Prozedur zunächst eine Addition von 0.0 durch, so daß der Fließkomma-Akkumulator belegt wird. Anschließend wird die Routine aufgerufen. Sie speichert den String ab der Adresse 256 und schließt ihn mit dem Zeichen CHR(0) ab. Diesen String kopiert die Prozedur in die Stringvariable S, wobei der String mit Leerstellen zur vollen Länge erweitert wird.

```
PROGRAM KONVERT(INPUT, OUTPUT);
  CONST MAXLEN = 20;
  TYPE STRING  = ARRAY[0..MAXLEN] OF CHAR;
  VAR I: INTEGER;
      R: REAL;
      S: STRING;

PROCEDURE REALTOSTRING(R:REAL; VAR S:STRING);
  CONST FLPSTR=-16931; (* $BDDD *)
        BUF   = 256;   (* $0100 *)
  VAR I: INTEGER;
BEGIN
 R:=R+0.0;  (* Fließkomma-Akummulator belegen *)
 SYS(FLPSTR);   (* Umwandlung *)
 I:=0;
 WHILE PEEK(BUF+I)<>0 DO
  BEGIN
   S[I]:=CHR(PEEK(BUF+I)); I:=I+1
  END;
 WHILE I<=MAXLEN DO
  BEGIN
   S[I]:=" "; I:=I+1
  END;
END; (* REALTOSTRING *)
BEGIN
 READLN(R);
 REALTOSTRING(R,S);
 FOR I:=0 TO MAXLEN DO
  WRITE(S[I]:2);          (* drucke mit Lücken *)
END.
```

3.2 Tips zum Editor

Sie können den Editor auch ohne den Pascal-Compiler benutzen. Vor dem Aufruf des Editors mit SYS 32768 geben Sie den Speicherbereich an, der für den Text zur Verfügung steht.

```
641/642  L und H-Byte Textanfang
643/644  L und H-Byte Textende
```

Da der Editor den Speicherbereich von $7F00 bis $A000 benötigt, können Sie mit den folgenden Befehlen den gesamten restlichen Speicher für den Text freigeben:

```
POKE 641,0: POKE 642,8:
POKE 643,0: POKE 644,127:
SYS 32768
```

Bei der Rückkehr vom Editor sind die Z-Page-Zeiger für BASIC von 43 bis 56 unverändert.

Sollten Sie auch die Dokumentation der Pascal-Programme mit dem Editor erstellen, so können Sie mit dem Befehl CHANGE direkt Steuerzeichen für den Drucker in den Text einfügen. Wollen Sie z.B. nach der Zeile 60 einen Seitenvorschub erzeugen, so können Sie in Zeile 61 folgende Zeichen eingeben.

```
&%
```

Mit CHANGE & #147 und CHANGE % #12 werden diese Zeichen in Kontrollzeichen umgewandelt, die von der Tastatur nicht direkt erreichbar sind. Geben Sie später den Text mit OUT an den Drucker aus (siehe Abschnitt 4.2.11), so wird der Drucker (MPS-802) nach Zeile 60 einen Seitenvorschub (skip page) ausführen.

Besonders bei der Erstellung von Tabellen sind die variablen Textgrenzen und die Line-Commands O und OO sinnvoll anzuwenden: Möchten Sie um eine Tabelle einen Rand aus Ausfrufezeichen "!" legen, könnten Sie wie folgt vorgehen:

```
!          !          !                          !
Variable   Adresse    Bedeutung
RSMEFLG    908        >0, falls RESUME erlaubt
PNT        909-922    Basic-Zeiger 43-56
COLOR-B    8957       Hintergrundfarbe
COLOR-F    8952       Rahmenfarbe
COLOR-T    8994       Farbe Textfenster
COLOR-H    8974       Farbe Kopfzeile
COLOR-L    9003       Farbe Zeilennummern
```

Die erste Zeile enthält die Markierung, die *über* die darunterstehende Tabelle kopiert werden soll. Deshalb geben Sie in der Zeile mit den Ausrufezeichen das Line-Command C ein. Außerdem markieren Sie die erste und letzte Zeile der Tabelle mit OO. Dadurch werden die Ausrufezeichen in die Tabelle kopiert. Wenn Sie ein wenig mit den Textgrenzen in der Zeile BND= experimentieren, werden Sie auch einen Weg finden, selektiv einzelne Spaltenbereiche mit),)), (und ((zu verschieben oder zu löschen.

Die obige Tabelle zeigt übrigens einige globale Variablen des Pascal-Systems. So könnten Sie z.B. in BASIC mit

```
POKE 908, 1
```

die Eingabe eines Fragezeichens beim nächsten Aufruf des Pascal-Menüs zulassen. Falls Sie die angegebenen Farben im Pascal-System ändern wollen, müssen Sie direkt nach dem Laden des Systems (vor RUN) die entsprechenden Speicherzellen mit POKE von Basic aus verändern. Wenn Sie das Pascal-System anschließend speichern, sind die Änderungen permanent.

4 Dokumentation Pascal-System

Inhalt der Systemdiskette

PASCAL-SYSTEM	Compiler, Editor und Laufzeitsystem.
MERGE.P	Beispiel für Include-Files und Dateien (benötigt bei Übersetzung FILE.INC).
FILE.INC	Include-File, definiert die Prozeduren RESET und REWRITE (siehe Abschnitt 4.4.5.1 und 2.16).
RELATIV.P	Demonstration für Relativdateien mit C 64.
QUEEN.P	Problem der acht Königinnen.
TREE.P	Darstellung einer Baumstruktur auf dem Drucker.
ROMBERG.P	Mathematikprogramm. Numerische Berechnung von Integralen.

Start des Systems

- Zunächst müssen Sie alle Erweiterungsmodule abschalten, da diese eventuell die Funktion des Pascal-Systems stören könnten.

- Laden Sie jetzt das Programm PASCAL-SYSTEM von der Diskette.

- Nach dem Start des Programmes mit RUN erscheint das Pascal-Menü, von dem aus Sie Programme erstellen, übersetzen und testen können.

Allgemeines

Alle Eingaben im System sind so organisiert, daß Sie mit möglichst wenigen Zwischenschritten jede Funktion erreichen können. Dabei besitzen im allgemeinen die Zeichen '*' und '?' eine Sonderfunktion. Alle Eingaben bei blinkendem Cursor müssen mit RETURN beendet werden. Grundsätzlich wird bei längeren Operationen (Laden, Speichern) eine Abfrage der RUN-STOP-Taste vorgenommen, so daß die Ausführung jederzeit abgebrochen werden kann.

4.1 Das Pascal-Menü

```
PASCAL-MENU
-----------

SELECT OPTION:

NAME  EDIT NEW DATASET
'?'   RESUME EDIT
'$'   COMPILE DATASET
'*'   EXIT TO BASIC

==>
```

Eingabe eines Namens (max. 16 Zeichen):

Der Editor sucht einen Text auf der Diskette im Laufwerk 0 mit der Geräteadresse 8. Als Typ wird das Suffix ',PRG' benutzt. Tritt beim Ladevorgang ein Fehler auf, so springt der Editor zum Pascal-Menü zurück. Überprüfen Sie, ob die Floppy betriebsbereit war, und wiederholen Sie die Eingabe.

Konnte der angegebene Text auf der eingelegten Diskette nicht gefunden werden, so nimmt der Editor an, daß Sie einen neuen Text mit diesem Namen anlegen möchten. Es erscheint die folgende Meldung:

```
THIS IS A NEW DATASET!

ENTER RECORD LENGTH:

[RANGE:1.80]
['*' OR '?' FOR END]

==>
```

An dieser Stelle bestimmen Sie die maximale Länge einer Textzeile für den neuen Datenbestand. Gültige Werte liegen im Bereich zwischen 1 und 80.

Wollen Sie jedoch keinen neuen Datenbestand anlegen, so können Sie mit '*' oder '?' zum Pascal-Menü zurückkehren.

Eingabe von '?'

Um den Datenbestand zu editieren, der sich bereits im Speicher befindet, genügt diese Eingabe. Sie ersparen sich hiermit die Ladezeit von der Diskette.

Eingabe von '$'

Es wird der Compiler aufgerufen, der den momentan im Speicher stehenden Datenbestand übersetzt (siehe Abschnitt 4.3).

Eingabe von '*'

Sie verlassen mit diesem Befehl das Pascal-System und kehren nach BASIC zurück (siehe Abschnitt 4.3.4). Von dort aus wird das Pascal-Menü durch die Eingabe von '*' erreicht.

4.2 Der Editor

4.2.1 Allgemeines

Mit dem Programm werden Texte im Arbeitsspeicher des Computers bearbeitet. Eingaben erfolgen über ein *Textfenster*, das in allen vier Richtungen (wie über ein Blatt Papier) mit den Funktionstasten verschoben werden kann. Innerhalb des Pascal-Systems stehen nur 8 Kbyte Textspeicher zur Verfügung.

Durch den Einschluß von Programmtexten von der Diskette können jedoch beliebig große Programme modular erstellt werden. Einzelheiten werden in Abschnitt 4.4.6.2 (Include-Files) erklärt.

Die erstellten Texte werden auf Diskette gespeichert, wobei neben dem Text selbst auch Informationen über Tabulatoren etc. gespeichert werden, so daß Sie sich Textbausteine erstellen können.

4.2.2 Gliederung des Bildschirms

```
                                        Scroll-Betrag

  Kopf-Zeile--> 1.COL:0001                SCROLL:HALF
                MSK=
      Status--> BND=<
                TAB=
                COL=0----+----1----+----2----+----3----+
   TOP-Zeile-->     ************* TOP ******************
                0000
                0001
                0002
                0003          Text-Bereich
                0004
                0005
                0006
BOTTOM-Zeile-->     ************ BOTTOM ****************
```

Das Bild gliedert sich in verschiedene Bereiche, die jeweils spezielle Aufgaben besitzen:

a) Die Kopfzeile (weiß)

In der Kopfzeile werden (Fehler-)Meldungen des Editors angezeigt. Liegen keine Meldungen vor, so wird die Nummer der ersten Textspalte im Textfenster angezeigt.

Andererseits werden in dieser Zeile auch Befehle, die sogenannten Primary-Commands, eingegeben. Bei der Eingabe in diese Zeile wird die zuvor angezeigte Meldung ausgeblendet.

b) Der Scroll-Betrag (weiß)

Beim Blättern mit den Funktionstasten verschiebt sich der Bildschirm um eine Anzahl von Zeilen oder Spalten, die hier angezeigt wird. Sie können diesen Wert jederzeit durch Überschreiben ändern. Folgende Spezifikationen sind erlaubt:

HALF halbe Bildlänge und -breite
PAGE ganze Bildlänge und -breite
nnnn vierstellige Zahl (kleiner als 128)

Anfangswert = HALF

c) Die Statuszeilen

Diese Zeilen werden nur bei Bedarf (mit dem Primary-Command PROF) eingeblendet.

- Der Eintrag nach 'MSK=' bildet eine Maske, die beim Einfügen neuer Zeilen vorgegeben wird. Alle Zeichen sind erlaubt. Anfangswert = Leerzeile.

- Der Eintrag nach 'BND=' begrenzt den Spaltenbereich bei der Texteingabe. Auch die Befehle FIND und CHANGE werden auf diesen Spaltenbereich begrenzt. Gültige Zeichen, die jeweils genau einmal auftreten dürfen, sind:

  '<' markiert den linken Rand
  '>' markiert den rechten Rand

  Anfangswert = ganze Zeile.

- Der Eintrag nach 'TAB=' setzt Tabulatoren, die im Textmodus (siehe Abschnitt 4.2.6) mit der Taste SHIFT-RETURN angesprungen werden. Jedes Zeichen entspricht einem gesetzten Tabulator. Anfangswert = keine Tabulatoren.

Die obigen drei Statuszeilen lassen sich durch Überschreiben auf neue Werte setzen, die auf ihre Gültigkeit geprüft werden.

- Schließlich wird nach 'COL=' eine Spaltenmarkierung ausgegeben, die zur Orientierung im Text dient.

d) Die Zeilen TOP und BOTTOM

Diese beiden Zeilen kennzeichnen am Bildschirm den Anfang und das Ende des Textes. Besonders zu beachten ist, daß am linken Rand der TOP-Zeile das Line-Command I(nsert) möglich ist.

e) Die Zeilennummern (weiß)

Alle Textzeilen sind von 0000 bis 9999 durchnumeriert. In diesem Bereich werden die Line-Commands eingegeben. Die Zeilennummern dienen nur zur Orientierung und werden nicht mit dem Text gespeichert.

f) Das Textfenster

Rechts von den Zeilennummern beginnt das Textfenster, das nach oben durch die Kopfzeile oder die Zeile TOP begrenzt wird. Den unteren Rand bildet die letzte Bildschirmzeile oder die Zeile BOTTOM. Außerdem sind alle Spalten nach der letzten Textspalte für Eingaben von der Tastatur gesperrt.

4.2.3 Cursorsteuerung

Die Steuerung des Cursors erfolgt mit den folgenden Tasten, die - soweit möglich - die gleiche Funktion wie bei dem BASIC-Editor besitzen:

CRSR UP	Cursor eine Zeile höher. Am oberen Bildrand Sprung in die letzte Bildschirmzeile.
CRSR DWN	Cursor eine Zeile tiefer. Am unteren Bildrand Sprung in die erste Bildschirmzeile.
CRSR ->	Cursor eine Spalte nach rechts. Am rechten Bildrand Sprung in die erste Spalte der gleichen Zeile.
CRSR <-	Cursor eine Spalte nach links. Am linken Bildrand Sprung in die letzte Spalte der gleichen Zeile.
HOME	Sprung an die erste Position der Kopfzeile. Nochmalige Betätigung bewirkt einen Sprung zum Scroll-Betrag.
CLR	Löschen der Kopfzeile und Sprung an die 1. Position der Kopfzeile.

INSERT — Einschub eines Zeichens an der laufenden Cursorposition. Arbeitet nur im Textfenster. Der Text bis zur letzten Textspalte ('>' in Zeile 'BND=') wird verschoben. Ist am Zeilenende kein Platz mehr, so ist die INSERT-Taste gesperrt.

DELETE — Löschen des Zeichens links neben dem Cursor. Arbeitet nur im Textfenster. Der Text bis zur letzten Textspalte ('>' in Zeile 'BND=') wird verschoben.

RETURN — Wirkung abhängig vom Eingabemodus:
Im Textmodus Sprung zur 1.Textspalte ('<' in Zeile 'BND=') der folgenden Textzeile.
Sonst Sprung zur ersten Spalte in der nächsten Bildschirmzeile.

f1 — Scroll up => Textfenster nach oben.

f3 — Scroll down => Textfenster nach unten.

f5 — Scroll right => Textfenster nach rechts.

f7 — Scroll left => Textfenster nach links.

f2 — Wiederhole den letzten FIND-Befehl (siehe Abschnitt 4.2.9). Setze den Cursor auf die entsprechende Textposition.

f4 — Wiederhole den letzten CHANGE-Befehl (siehe Abschnitt 4.2.9). Setze den Cursor auf die entsprechende Textposition.

Zusätzlich gibt es noch die Taste SHIFT-RETURN. Mit ihr werden alle Befehle auf dem Bildschirm (Primary- und Line-Commands) gelesen und ausgeführt. Dies geschieht auch automatisch bei jedem Blättern. Der Cursor bleibt beim Blättern an seiner letzten Textposition, solange diese noch auf dem neuen Bild vorhanden ist. Ansonsten springt er in die linke obere Ecke des Bildschirmes.

Alle anderen Sondertasten (RVS ON, CTRL-BLK etc.) werden als invertierte Zeichen am Bildschirm dargestellt und auch in den Text eingefügt.

4.2.4 Primary-Commands

Die Eingabe erfolgt in der Kopfzeile. Schlüsselworte sind von eventuell folgenden Parametern durch mindestens ein Leerzeichen zu trennen. Die Befehle werden beim Blättern oder nach der Eingabe von SHIFT-RETURN ausgeführt.

Wird ein unbekannter Befehl eingegeben, so wird die gesamte Kopfzeile als Befehl an die Floppy geschickt. So löscht man z.B. mit S0:TEXT1 den Datenbestand TEXT1 im Laufwerk 0. Nach Ausführung des Befehls wird der Diskettenstatus im folgenden Format angezeigt:

Fehlernummer, Fehlertext, Spur, Sektor

Nähere Details entnehmen Sie bitte dem Handbuch der Floppy.

Tabelle Primary-Commands

Befehl	Kurzform	Wirkung
RESET	RES	Alle Statuszeilen ausblenden. Alle unvollständigen Line-Commands löschen. Textmodus verlassen. FIND- und CHANGE-Funktionstasten ausschalten. Repeat-Funktion aller Tasten ausschalten.
PROFILE	PROF	Alle Statuszeilen anzeigen.
MSKS	MSK	Zeile mit Einfügemaske anzeigen.
BNDS	BND	Zeile mit Textgrenzen anzeigen.
TABULATOR	TAB	Tabulatorzeile anzeigen.
COLUMNS	COL	Spaltenkennzeichnung anzeigen.
END	END	SAVE ausführen und Rückkehr zum Pascal-Menü.
CANCEL	CAN	Rückkehr zum Pascal-Menü. **Achtung!** Der Text wird nicht auf Diskette gesichert!

SAVE	SAVE	Alten Text unter diesem Namen auf der Diskette löschen. Neuen Text speichern. Wiederholung dieser Schritte, bis kein Verify-Error mit dem Original im Speicher auftritt.
TEXT	TE	Textmodus einschalten (siehe Abschnitt 4.2.6).
REPEAT	REP	Alle Tasten erhalten Repeat-Funktion.
LOCATE n	L n	Zeile n im Text aufsuchen (n = 0000 ..9999). Fehlt n, erfolgt ein Sprung zum Textanfang.
FIND	F	Suche nach Zeichenfolge (siehe Abschnitt 4.2.7).
CHANGE	C	Ersetzen von Zeichenfolgen (siehe Abschnitt 4.2.8).
INPUT	IN	Einlesen einer sequentiellen Datei von einem Peripheriegerät (siehe Abschnitt 4.2.10).
OUTPUT	OUT	Ausgabe des Textes im Speicher als sequentielle Datei (siehe Abschnitt 4.2.11).
COPY	COPY	Kopieren von Teilen aus Texten auf der Diskette (siehe Abschnitt 4.2.12).

4.2.5 Line-Commands

Zur Texteditierung stehen die folgenden wirkungsvollen Zeilen- und Blockbefehle zur Verfügung. Sie werden im Zeilennummernbereich oder links in der Zeile TOP eingegeben und beim Blättern oder der Eingabe von SHIFT-RETURN ausgeführt.

Falls eine Zahl (n) angegeben werden kann, so wird eine fehlende Zahl durch 1 ersetzt (I entspricht also I1). Gültige Werte für n liegen zwischen 1 und 128.

Blockkommandos müssen in zwei Zeilen eingegeben werden. Sie kennzeichnen den Anfang und das Ende des zu bearbeitenden Blockes. Ein Block darf nicht mehr als 256 Zeilen umfassen.

Beispiele

I8 in Zeile TOP fügt am Textanfang acht Leerzeilen ein.

RR in Zeile 1 und in Zeile 4 wiederholt alle Zeilen zwischen 1 und 4 einmal.

I(nsert)	n	Nach dieser Zeile werden n Zeilen eingefügt.
D(elete)		Diese Zeile wird gelöscht.
DD		Ein Block beginnend ab der ersten DD-Zeile bis zur zweiten DD-Zeile (einschließlich) wird gelöscht.
R(epeat)	n	Diese Zeile wird n-mal wiederholt.
RR	n	Ein Block wird n-mal wiederholt.
>	n	Diese Zeile wird um n Spalten nach rechts geschoben, falls dadurch keine Zeichen außer dem Leerzeichen am rechten Rand verlorengehen. Der Spaltenbereich wird durch die Einträge '<' und '>' in der BND-Zeile festgelegt.
>>	n	Desgleichen mit einem Block von Zeilen.
<	n	Diese Zeile wird um n Spalten nach links geschoben, falls dadurch keine Zeichen außer dem Leerzeichen verlorengehen.
<<	n	Desgleichen mit einem Block von Zeilen.
)	n	Diese Zeile wird um n Spalten nach rechts geschoben. Dabei können eventuell am rechten Rand Zeichen herausgeschoben werden.
))	n	Desgleichen mit einem Block von Zeilen.
(	n	Analog zu ')' nach links verschieben.

((	n	Desgleichen mit einem Block von Zeilen.
M(ove)		Kennzeichnet eine Zeile, die an eine neue Position gestellt werden soll.
MM		Markiert den Anfang und das Ende eines Blockes, der verschoben werden soll.
C(opy)		Diese Zeile soll kopiert werden.
CC		Ein Block von Zeilen wird zum Kopieren markiert.

Bei M und C muß noch eine Zielposition angegeben werden:

A(fter)	Der Block wird hinter diese Zeile gestellt.
B(efore)	Der Block wird vor diese Zeile gestellt.
O(verlay)	Die Zeile wird *über* diese Zeile kopiert, so daß Leerstellen in der O-Zeile durch Zeichen der M- oder C-Zeile ersetzt werden. Der Kopiervorgang ist dabei auf den Spaltenbereich zwischen '<' und '>' in der BND-Zeile begrenzt.
OO	Der Block wird *über* den Block zwischen den beiden OO-Zeilen kopiert. Ist die Anzahl der zu kopierenden Zeilen kleiner als der Zielblock, werden die Zeilen zyklisch wiederholt.

4.2.6 Textmodus

Der Textmodus ist beim Aufruf des Editors eingeschaltet. Nachdem er mit RESET ausgeschaltet wurde, wird er mit TEXT wieder aktiviert. Dieser spezielle Eingabemodus dient zur Eingabe langer, zusammenhängender Texte. Erreicht nämlich der Cursor bei der Eingabe von Zeichen den rechten Bildrand, so folgt das Textfenster automatisch (um den gewählten Scroll-Betrag) dem Cursor. Beim Erreichen des rechten Textrandes springt der Cursor an den linken Textrand der nächsten Zeile. Durch die Eingabe von SHIFT-RETURN springt der Cursor an die nächste vortabulierte Position ('-' in der Zeile 'TAB=').

4.2.7 FIND

Mit dem FIND-Befehl können Sie nach Zeichenfolgen (Strings) mit bis zu 32 Zeichen Länge im Text suchen. Der Befehl kann um Parameter erweitert werden, die in der folgenden Reihenfolge auftreten. Dabei stellen übereinanderstehende Parameter eine Auswahl dar, von der nur eine Variante pro Befehl angegeben werden kann.

```
        String   ALL     CHARS
FIND    'String' FIRST   WORD        nnn
        #kkk     NEXT    PREFIX    nnn-mmm
          *              SUFFIX
```

String — Zeichenfolge von maximal 32 Zeichen ohne Leerzeichen. Die Zeichenfolge darf nicht mit *, # oder ' beginnen.

'String' — Die Zeichenfolge darf aus 32 beliebigen Zeichen außer ' bestehen.

* — Es wird der Suchstring des letzten FIND- oder CHANGE-Befehls benutzt.

#kkk — Es wird nach dem ASCII-Zeichen mit dem dezimalen Code nnn gesucht (z.B. #13 ist CR).

NEXT — Die Suche beginnt ab der augenblicklichen Cursorposition im Textfenster. Steht der Cursor nicht im Textfenster, so wird ab dem linken Textrand der 1. Bildschirmzeile gesucht.

FIRST — Die Suche beginnt bei der Textzeile 0000.

ALL — Es werden alle Strings im gesamten Text gezählt.

CHARS — Der String wird auch als Teil eines anderen Wortes gefunden (z.B FIND *der* CHARS findet auch das Wort *jeder*).

PREFIX — Der String muß einem Trennzeichen folgen.

SUFFIX — Dem String muß ein Trennzeichen folgen.

WORD — Der String muß mit einem Trennzeichen beginnen und enden.

nnn Der String muß in Spalte nnn beginnen.

nnn-mmm Der String muß zwischen Spalte nnn und mmm beginnen.

Nicht angegebene Parameter werden durch die Werte NEXT, CHARS und die momentan gültigen Textgrenzen aus der Statuszeile nach 'BND=' ersetzt. Außerdem sind die Abkürzungen FIR, NEX, WOR, PRE und SUF erlaubt. Nach Ausführung des Befehls steht der Cursor auf der gefundenen Textposition.

Beispiele

FIND I WORD	findet I, I1, I-22, 3-I, aber nicht SIN, IN, OMI. Angezeigt wird das erste Auftreten.
FIND * ALL	zählt, wie oft der letzte String (I) im Text auftritt.
F X FIR	findet das erste X im Text.

4.2.8 CHANGE

Mit dem CHANGE-Befehl können Sie Zeichenfolgen (Strings) mit bis zu 32 Zeichen Länge im Text durch andere Zeichenfolgen mit ebenfalls maximal 32 Zeichen Länge ersetzen.

```
         String1   String2   ALL     CHARS
CHANGE   'String1' 'String2' FIRST   WORD        nnn
         #kkk      #kkk      NEXT    PREFIX    nnn-mmm
           *         *               SUFFIX
```

Die Suche nach String 1 wird wie beim FIND-Befehl durchgeführt und durch die Parameter gesteuert. Ist String 2 länger als String 1, so wird vor der Umwandlung geprüft, ob am Zeilenende ('>' in der Statuszeile 'BND=') genug Platz ist. Ist dies nicht der Fall, so wird eine Fehlermeldung ausgegeben. Der zweite String kann auch leer ('') sein, so daß String 1 vollständig gelöscht wird. Der Parameter ALL bewirkt die Umwandlung aller Strings im Text. Oft ist es jedoch sicherer, wie in Abschnitt 4.2.9 beschrieben mit den Funktionstasten f2 und f4 den Text durchzugehen.

Beispiel

CHANGE XXX " ALL WORD	löscht alle Worte XXX im Text.
CHANGE INTEGER REAL WOR	wandelt alle Zeichenfolgen INTEGER in REAL um.

4.2.9 Die Tasten f2 und f4

Um die ständige Wiederholung von FIND- und CHANGE-Befehlen zu vermeiden, kann der letzte FIND-Befehl mit f2 und der letzte CHANGE-Befehl mit f4 wiederholt werden. Genauer gelten folgende Regeln:

f2 entspricht FIND 'String1' NEXT. String 1 ist der letzte benutzte FIND- oder CHANGE-String. Die übrigen Parameter (z.B. WORD) entsprechen ebenfalls denen des letzten Befehls. Wird das Textende erreicht, so erscheint die Meldung **BOTTOM READCHED**. Die nächste Betätigung von f2 entspricht dann FIND 'String1' FIRST. Sollte der String nicht gefunden werden, wird STRING NOT FOUND angezeigt.

f4 entspricht CHANGE 'String1' 'String2' mit den letzten Strings und den zuletzt gültigen Parametern (wie WORD und nnn) bei CHANGE.

Beispiel

CHANGE Otto Karl FIRST.	Der Cursor springt auf das erste Wort Otto im Text und wandelt es in Karl um.
f2-Taste drücken	Der Cursor springt auf das nächste Wort Otto.
f4-Taste drücken	Der Cursor bleibt an der alten Position. Otto wird durch Karl ersetzt.
f2-Taste drücken	Der Cursor springt auf das nächste Wort Otto.
f2-Taste drücken	Es wird keine Änderung durchgeführt. Suche nach Otto.

etc.

4.2.10 INPUT

Mit diesem Befehl wird ein Menü aufgerufen, in dem nacheinander die folgenden Parameter festgelegt werden:

```
DEVICE NUMBER:
['*' OR '?' FOR END]
==>
SEK. ADDRESS:
['*' OR '?' FOR NONE]
==>
FILE NAME:
['*' OR '?' FOR NONE]
==>
CODE OF DELIMITER:
['*' OR '?' FOR NONE]
==>13
```

Die Parameter Geräteadresse, Sekundäradresse und Filenamen müssen, wie im Handbuch des Rechners beschrieben, angegeben werden. Mit diesen Parametern wird dann ein OPEN-Befehl (wie in BASIC und Pascal) ausgeführt.

Anschließend wird bis zum Dateiende von der Datei gelesen. Die eingelesenen Daten werden im Text eingefügt. Die Einfügeposition kann vor dem Aufruf von INPUT mit den Line-Commands A(fter) oder B(efore) festgelegt werden. Fehlt diese Angabe, so werden die Daten am Textanfang eingefügt.

Die Formatierung der eingelesenen Daten in Zeilen erfolgt durch die Angabe des Begrenzungszeichens **(Delimiter)**.

a) Kein Begrenzungszeichen

Beträgt die Satzlänge n Zeichen (z.B. n=50), so werden jeweils n Zeichen in jede Zeile gestellt, bevor eine neue Zeile eingefügt wird. Alle ASCII-Zeichen werden unverändert übernommen.

b) Mit Begrenzungszeichen

Normalerweise sind Textfiles (für BASIC und Pascal) durch das Zeichen Carriage Return (CR) mit dem ASCII-Code 13 in einzelne Zeilen gegliedert. Durch die Eingabe von 13 als Delimiter werden so lange Zeichen in eine Zeile geschrieben, bis das Begrenzungszeichen CR gelesen wird. Dieses Zeichen wird nicht in den Text eingefügt. Alle folgenden Zeichen werden in die nächste Zeile geschrieben. Somit bleibt beim Einle-

sen die Zeilenstruktur erhalten. Natürlich sind beliebige Begrenzungszeichen erlaubt.

Tritt ein Systemfehler auf, so wird das Einlesen beendet und ein Fehlercode in der Kopfzeile angezeigt.

Beispiel

INPUT mit folgenden Parametern:

```
DEVICE NUMBER:     8
SEK. ADDRESS:      0
FILE NAME:         $
CODE OF DELIMITER: 0
```

Hierdurch wird ab der markierten Zeile das Inhaltsverzeichnis der Diskette in der Codierung als BASIC-Programm eingelesen (entspricht also LOAD"$",8 in BASIC).

Beispiel

INPUT mit folgenden Parametern:

```
DEVICE NUMBER:     8
SEK. ADDRESS:      3
FILE NAME:         TEST,S,R
CODE OF DELIMITER: 13
```

Eine sequentielle Datei 'TEST,SEQ' auf der Diskette wird gelesen. Eine Anwendung des Befehls INPUT ist das Einlesen von Dateien, die mit anderen Programmen (z.B. Editoren) erstellt wurden. Speziell können so Textfiles gelesen werden, die in Pascal-Programmen mit WRITE erzeugt wurden.

4.2.11 OUTPUT

Mit diesem Befehl wird ein Menü aufgerufen, in dem nacheinander die folgenden Parameter festgelegt werden:

```
DEVICE NUMBER:
['*' OR '?' FOR END]
==>
SEK. ADDRESS:
['*' OR '?' FOR NONE]
==>
FILE NAME:
['*' OR '?' FOR NONE]
==>
CODE OF DELIMITER:
['*' OR '?' FOR NONE]
==>13
TRUNCATE TRAILING
SPACES? [YES OR NO]
==>Y
```

Die Wahl der Parameter Geräteadresse, Sekundäradresse und Filenamen erfolgt wie im Handbuch des Rechners beschrieben.

Mit diesen Parametern wird ein OPEN-Befehl (wie in BASIC) ausgeführt. Anschließend wird der gesamte Text im Arbeitsspeicher auf die Datei ausgegeben.

Die Formatierung kann durch die Angabe eines Begrenzungszeichens gesteuert werden:

a) Kein Begrenzungszeichen

Alle Zeichen einer Zeile werden ausgegeben. Ohne jegliches Trennzeichen folgen die Zeichen der nächsten Zeile.

b) Mit Begrenzungszeichen

Jede ausgegebene Zeile wird mit dem ASCII-Zeichen beendet, dessen Code als Delimiter angegeben wurde. BASIC- und Pascal-Programme erwarten das ASCII-Zeichen CR mit dem Code 13 am Ende jeder Zeile. Natürlich sind auch andere Trennzeichen möglich.

Oft ist es wünschenswert, daß Leerzeichen am Zeilenende abgeschnitten werden. Dies wird durch die Eingabe von 'Y' beim letzten Menüpunkt erreicht. Die Druckerausgabe kann z.B. durch dieses Abschneiden von nachlaufenden Leerzeichen in jeder Zeile erheblich beschleunigt werden.

Die Wahl eines Begrenzungszeichens und das Abschneiden sind beliebig kombinierbar.

Tritt bei der Ausgabe ein Systemfehler auf, so wird die Ausgabe beendet und ein Fehlercode in der Kopfzeile angezeigt.

Beispiel

OUTPUT mit folgenden Parametern:

```
DEVICE NUMBER:     4
SEK. ADDRESS:      0
FILE NAME:         *
CODE OF DELIMITER: 13
TRUNCATE SPACES:   Y
```

Hierdurch wird der gesamte Text auf den Drucker mit der Geräteadresse 4 und der Sekundäradresse 0 ausgegeben, wobei jede Zeile mit CR beendet wird.

Beispiel

OUTPUT mit folgenden Parametern:

```
DEVICE NUMBER:     8
SEK. ADDRESS:      3
FILE NAME:         TEST,S,W
CODE OF DELIMITER: 13
TRUNCATE SPACES:   Y
```

Der gesamte Text wird als Datei 'TEST,SEQ' auf Diskette gespeichert. Viele Programme (Assembler, Mailboxprogramme) können solche Dateien als Eingabe verwenden. Besonders nützlich ist diese Option auch zur Erzeugung von Testfiles für Pascal-Programme, die Textfiles mit READ und READLN lesen.

4.2.12 COPY

Mit diesem Befehl können Sie aus einem Editortext auf der Diskette Zeilen in den Arbeitsspeicher kopieren. Für diese Operation ist der Befehl INPUT nicht geeignet, da Editortexte in einem speziellen Format gespeichert werden.

Den Namen des Textes auf Diskette geben Sie in einem gesonderten Menü ein:

```
COPY DATASET
------------

ENTER DATASET-NAME:
['*' OR '?' FOR END]

==>
```

Wird der Text nicht gefunden, so erfolgt ein Rücksprung in den Editor. Dies geschieht ebenfalls bei der Eingabe von '*' oder '?'.

Problematisch ist das Kopieren aus einem Datenbestand, der eine andere Satzlänge besitzt. Hierbei können Zeilen zerstückelt oder zusätzliche Leerzeichen angefügt werden. Sie müssen deshalb das Kopieren durch die Eingabe von 'C' bestätigen:

```
THIS DATASET HAS A
DIFFERENT RECORD-SIZE!

CONFIRM COPY WITH 'C'!

==>
```

Jede andere Eingabe führt zum Editor zurück. Anschließend können Sie noch den Zeilenbereich festlegen, der kopiert werden soll.

```
FIRST LINE COPIED:
['*' OR '?' TO COPY ALL]
==>

LAST LINE COPIED:
==>
```

Ein '?' oder '*' bei der Eingabe der letzten Zeilennummer erlaubt Ihnen, die Anfangszeilennummer zu korrigieren. Um bis zum Textende zu kopieren, müssen Sie nur eine genügend große Zeilennummer eingeben (z.B. 9999).

Die eingelesenen Zeilen werden in den Text eingefügt. Die Einfügeposition kann vor dem Aufruf von COPY mit den Line-Commands A(fter) oder B(efore) festgelegt werden. Fehlt diese Angabe, so werden die Zeilen am Textanfang eingefügt.

4.2.13 Fehlermeldungen im Editor

Meldung	Bedeutung und Korrekturmöglichkeit
LINE-COMM. IGNORED	Es wurden zu viele Line-Commands von einem Typ angegeben. Die erkannten Line-Commands werden angezeigt (eventuell mit RESET die Line-Commands löschen).
COMMAND CONFLICT	Bei MOVE oder COPY liegt die A- oder B-Eintragung in dem zu kopierenden Block.
OUT OF MEMORY	Der Arbeitsspeicher ist zu klein, um den Text zu erweitern (in Pascal Include-Files benutzen).
BLOCK TOO LONG	Ein Block darf nicht mehr als 256 Zeilen umfassen (Block in einzelnen Teilen bearbeiten).
ILLEGAL BOUNDS	Ungültige Einträge in 'BND=' (siehe Abschnitt 4.2.2).
MOVE ERROR	Bei den Line-Commands '>' oder '<' würden Zeichen über den Textrand verschoben. Der Fehler tritt auch bei CHANGE auf, falls in der Einfügezeile nicht genügend Platz ist.
ILLEGAL COMMAND	Ungültige Parameter bei FIND und CHANGE.
ENTER A STRING!	Ein String bei FIND oder CHANGE hat nicht die korrekte Form.
ILLEGAL COLUMN	Eine Spaltenangabe bei FIND und CHANGE verläßt den Textbereich.
STRING FOUND	Meldung bei FIND. Cursor steht auf dem Suchstring.

BOTTOM REACHED	Bei FIND oder CHANGE wurde das Textende erreicht.
STRING NOT FOUND	Der Suchstring befindet sich nicht im Bereich, der durch die Textgrenzen bei 'BND=' festgelegt wird.
KEY NOT ACTIVE	Die f2- und f4-Funktionstasten sind nur aktiv, falls zuvor ein FIND- oder CHANGE-Befehl eingegeben wurde.
nnnn TIMES FOUND	Kein Fehler: Meldung nach FIND ALL.
nnnn CHANGED	Anzahl der geänderten Strings bei CHANGE (ALL)
nnnn/mmmm ERRORS	Bei CHANGE ALL wurden nnnn Strings gefunden, von denen mmmm nicht geändert werden konnten.
SYSTEM ERROR NR. n	Bei einem Betriebssystemaufruf trat ein Fehler auf: n=1 TOO MANY FILES OPEN n=2 FILE OPEN n=3 FILE NOT OPEN n=4 FILE NOT FOUND n=5 DEVICE NOT PRESENT n=6 NOT INPUT FILE n=7 NOT OUTPUT FILE n=8 MISSING FILE-NAME n=9 ILLEGAL DEVICE-NUMBER

4.3 Bedienung des Compilers

4.3.1 Wahl der Optionen

Grundsätzlich übersetzt der Compiler das Pascal-Programm, das sich augenblicklich im Speicher befindet. Eine Ausnahme hiervon bilden die Include-Files (siehe Abschnitt 4.4.6.2). Mit ihnen ist es möglich, beliebig große Teile des Programmes von der Diskette zu lesen und gemeinsam mit dem Programm im Speicher zu übersetzen.

Der Compiler meldet sich beim Aufruf mit der folgenden Meldung:

```
PASCAL 1.4
----------

SELECT OPTION:

0    CHECK SYNTAX
1    GENERATE CODE
ELSE LOCATE ADDRESS

==>1
```

An dieser Stelle wählen Sie den Übersetzungsmodus:

Eingabe 0 (Syntax-Prüfung):

Bei dieser Wahl prüft der Compiler den Quelltext auf syntaktische Korrektheit. Fehler werden im Programmlisting markiert. Es wird jedoch kein Code erzeugt.

Eingabe 1 (Code-Erzeugung):

Dieser voreingestellte Modus liest den Programmtext, prüft ihn auf Korrektheit und erstellt im Speicher ein ablauffähiges Programm.

Überschreitet die Länge des erzeugten Objektprogrammes etwa 11 Kbyte, so erstellt der Compiler zwei temporäre Dateien (C$ und F$). Am Ende der Übersetzung werden diese Dateien gelesen und im Speicher abgelegt, so daß dann ebenfalls der Code komplett im Speicher steht.

Eingabe einer anderen Zahl (Finde Adresse):

Mit dieser Option können Sie zu einer Adresse im Code (Objektprogramm) die zugehörige Position im Pascal-Programm (Quelltext) lokalisieren. Treten z.B. bei der Ausführung des Objektprogrammes Fehler auf (Division durch 0 etc.), so wird eine Fehlermeldung mit der Adresse des fehlerhaften Befehls ausgegeben. Geben Sie nun diese Fehleradresse an dieser Stelle an, so wird die Quelltextposition des Divisionsbefehls im Listing mit der Fehlernummer 0 markiert.

Anschließend werden Sie nach der Startadresse des Objektprogrammes gefragt:

```
P-CODE START:
==>
```

In 99,9 Prozent aller Fälle werden Sie hier nur RETURN eingeben, da eine Änderung der vorgegebenen Anfangsadresse ohne weitere Maßnahmen kein lauffähiges Programm erzeugt. Diese Maßnahmen werden in Abschnitt 4.3.3 beschrieben.

Schließlich können Sie bei Bedarf das Programmlisting mit den Fehlermeldungen auf den Drucker umleiten, indem Sie bei der folgenden Abfrage ein anderes Zeichen als 'N' (für NO) eingeben.

```
LISTING TO PRINTER?
==>N
```

4.3.2 Meldungen im Compiler

Bei der Übersetzung eines jeden Programmblockes wird einmal geprüft, ob die STOP-Taste betätigt wurde. Ist dies der Fall, so wird folgende Bestätigungsmeldung ausgegeben:

```
'*' STOP  '?' SYNTAX
ELSE CONTINUE
==>
```

'*' bricht die Übersetzung ab. Natürlich steht dann kein vollständiges Objektprogramm im Speicher.

'?' schaltet auf den Modus Syntax-Test um. Der Programmtext wird nur noch auf syntaktische Korrektheit geprüft, ohne daß dabei Code erzeugt wird. Ist im Augenblick eine temporäre Datei eröffnet, so wird diese Datei geschlossen.

Jede andere Eingabe setzt die Übersetzung im bisherigen Modus fort.

Wird ein **Fehler** im Quellprogramm gefunden, so markiert ein Pfeil das erste Zeichen nach dem Symbol, bei dessen Verarbeitung der Fehler entdeckt wurde.

Beispiel (Semikolon fehlt):

```
 BEGIN X:=Y;Y:=Z Z:=X END
                 ↑
***** ERROR 14 IN xxxx
```

xxxx ist dabei die Zeilennummer im Quelltext. Wird Text von einem Include-File gelesen, so bezieht sich die Zeilennummer auf die Zeilen in diesem File.

Die Fehlernummer (hier also 14) wird im Anhang B erklärt.

Anschließend wird die obige Bestätigungsmeldung angezeigt. Wurde eine Fehleradresse gesucht (Fehler 0), so beendet der Compiler die Arbeit und kehrt zum Pascal-Menü zurück.

4.3.3 Spezielles

Wenn Sie bereits längere Zeit mit dem Compiler gearbeitet haben und die Möglichkeiten des Pascal-Systems voll ausnutzen wollen, dann sind eventuell die folgenden Informationen relevant:

Prinzipiell können Programme den gesamten BASIC-Arbeitsspeicher belegen. Da der Compiler *verdeckten* Speicher des C 64 benutzt, gibt es von dieser Seite auch keine Probleme. Jedoch steht der Quelltext im Bereich von $6000 bis $8000. Außerdem belegt der Editor den Speicherbereich von $8000 bis $A000. Deshalb prüft der Compiler bei der Rückkehr, ob diese Speicherbereiche durch das Objektprogramm überschrieben wurden. Ist der Quelltext überschrieben worden, was durch die Benutzung von Include-Files unproblematisch ist, wird die Option RESUME im Pascal-Menü gesperrt. Ist außerdem der Editor vom Code überschrieben worden, kehrt der Compiler nicht zum Pascal-Menü zurück, sondern verläßt das Pascal-System nach BASIC. Dort sollten Sie dann natürlich nicht versuchen, mit '*' das System neu zu starten.

Um nun dem übersetzten Programm den Speicher ab $6000 zur Verfügung zu stellen, geben Sie folgende BASIC-Befehle ein:

```
POKE 55,0: POKE 56,160: CLR
```

Es ist jedoch sehr unwahrscheinlich, daß Sie jemals soviel Speicherplatz benötigen werden, da zum Beispiel der komplette Compiler Pascal 1.4, der selbst in Pascal geschrieben ist, *nur* etwa 20 Kbyte belegt.

Außerdem besitzt das Pascal-System eine gewisse Flexibilität, die eine einfache Zusammenarbeit z.B. mit Maschinenprogrammen erlaubt. Ein übersetztes Programm besteht aus zwei Teilen:

1. Das Laufzeitsystem mit Hilfsprogrammen von 2049 bis 5611.
2. Der eigentliche Objektcode ab 5611.

Das Laufzeitsystem erwartet nun den Objektcode ab der Adresse 5611. Jedoch läßt sich der Code auch ab einer anderen Adresse ablegen (siehe Compiler-Menü). Überdies kann noch der Stapelspeicher des übersetzten Programmes beliebig verschoben werden. Um dem Laufzeitsystem die neuen Adressen mitzuteilen, müssen folgende Bytes gesetzt werden:

```
Adresse 5611 <= Stackanfang L-Byte
Adresse 5612 <= Stackanfang H-Byte
Adresse 5613 <= 18 (dezimal)
Adresse 5614 <= (P-CODE Anfang+2) L-Byte
Adresse 5615 <= (P-CODE Anfang+2) H-Byte
```

Normalerweise beginnt der Stapelspeicher (Stack) direkt hinter dem Objektprogramm. Er wächst nach *oben*, d.h zu steigenden RAM-Adressen. Außerdem existiert noch ein Bereich für dynamische Variablen (HEAP). Dieser Speicherbereich wächst ab dem Speicherende für BASIC nach *unten*. Beim Zusammenstoß zwischen HEAP und STACK wird eine Fehlermeldung vom Laufzeitsystem erzeugt.

Beispiel

Bei Übersetzung:

```
P-CODE-START = 9000
```

In BASIC:

```
POKE 5611, PEEK(9000)
POKE 5612, PEEK(9001)
POKE 5613, 18
POKE 5614, 9002 AND 255
POKE 5615, 9002 / 256
```

Durch diese Eingaben bleibt ein freier Speicherbereich zwischen 5616 und 9001, der z.B. durch Maschinenprogramme, Bildschirmspeicher etc. belegt werden kann. Der Code beginnt bei Adresse 9000 und endet an der vom Compiler am Schluß angezeigten Speicherstelle. Durch die ersten beiden

POKE-Befehle wächst der Stack (Zeiger 47/48) vom Code-Ende (Zeiger 5611/5612) aufwärts. Jeder Prozeduraufruf von NEW im Pascal-Programm läßt dann zur Laufzeit den Heap (Zeiger 59/60) vom Speicherende (Zeiger 55/56) nach *unten* wachsen.

4.3.4 Rückkehr zu BASIC

Nach der Rückkehr aus dem Pascal-Menü mit '*' befindet sich im Speicher das Laufzeitsystem für Pascal-Objektprogramme. (LIST probieren!). Dieses Programm darf nicht gelöscht, überschrieben oder geändert werden, da sonst das Pascal-System nicht korrekt arbeitet.

Falls Sie beim letzten Aufruf des Compilers mit '$' ein ausführbares Programm erstellt hatten, können Sie dieses wie ein *normales* BASIC-Programm behandeln:

Mit SAVE können Sie das Programm speichern und anschließend mit VERIFY das Programm auf der Diskette prüfen. Natürlich können die so gespeicherten Programme auf jedem C 64 geladen und ohne weitere Hilfsprogramme ausgeführt werden.

Bitte beachten Sie, daß Kassettenoperationen (im Editor oder in BASIC) Teile des Compilers überschreiben. Sollten Sie also ein Programm auf Kassette gespeichert haben, müssen Sie vor einer erneuten Übersetzung das Pascal-System neu laden.

Mit RUN können Sie das Programm testen. Eventuell auftretende Fehler werden im Klartext mit der Fehleradresse angezeigt. Außerdem erfolgt bei Funktionen und Prozeduren eine Auflistung der letzten Aufrufadressen (dynamic chain).

Ein Beispiel soll die Bedeutung dieser Verweiskette klären: Angenommen, Sie hätten ein Programm gestartet, das im Hauptprogramm HAUPT die Prozedur P1 aufruft. Diese Prozedur selbst benutzt die Funktion F1, in der eine Multiplikation 1000*1000 stattfindet. Diese Operation führt schließlich zu einer Überschreitung des Zahlenbereichs für INTEGER-Zahlen.

Dann wird folgende Fehlermeldung erzeugt:

```
INTEGER OVERFLOW        <- Fehlerursache
ERROR AT .....          <- Fehleradr. in Funktion F1
CALLED AT .....         <- Aufrufadr. in Prozedur P1
CALLED AT .....         <- Aufrufadr. im Hauptprogr.
```

Mit der Option LOCATE ADDRESS kann jede der angegebenen (Aufruf)-Adressen im Quellprogramm lokalisiert werden.

Das Pascal-Menü erreichen Sie von BASIC durch die Eingabe eines Sterns '*'. Von dort können Sie dann den Fehler im Quelltext mit dem Editor beheben, den Text compilieren und dann einen neuen Probelauf starten.

4.4 Sprachbeschreibung Pascal 1.4

4.4.1 Grundsätzliches

Der Compiler wurde entworfen, um eine möglichst vollständige und getreue Realisierung des *Standard*-Sprachumfangs zur Verfügung zu haben, der in Buch 1 (siehe Anhang E) beschrieben wird. Im folgenden wird diese Quelle als REPORT bezeichnet.

Außerdem sollte der erzeugte Code im Speicherplatzbedarf und in der Geschwindigkeit attraktiv im Verhältnis zum BASIC-Interpreter und zu compilierten BASIC-Programmen sein. Schließlich sollte der Aufwand für einen Zyklus Quelltextänderung, Übersetzung und Testlauf so gering wie möglich bleiben.

Daher wird der Compiler durch Banking *verdeckt* im Hintergrund gehalten und nur zur Übersetzung in den Vordergrund kopiert. Somit ist ein Zugriff auf die Floppy nur für Include-Files und zum Abspeichern der Programme nötig.

Um eine möglichst große Portabilität der Programme zu sichern, wurden bewußt keine rechnerspezifischen Sprachelemente (Grafik, Sound etc.) aufgenommen.

Andererseits wurden *Low-level*-Sprachelemente hinzugefügt, um in Pascal Speicheradressen zu adressieren und Bit-Operationen durchzuführen. Bei Bedarf können also die obigen Anwendungen explizit in Pascal programmiert werden.

4.4.2 Sprachumfang

Als Referenz für den Sprachumfang dienen die Syntax-Diagramme im Anhang A. In diesem Abschnitt werden alle semantischen Details angesprochen, die vom REPORT abweichen oder von diesem nur ungenau erfaßt werden.

Bezeichner

Sie dürfen beliebig lang sein (14 Zeichen signifikant), aber keine Buchstaben enthalten, die mit SHIFT auf der Tastatur eingegeben werden. Das sind in Abhängigkeit von der Bildschirmdarstellung Grafikzeichen oder Großbuchstaben.

Datentypen

Alle Datenstrukturen des Standards sind vorhanden. Die Objekte der einzelnen Typen benötigen den folgenden Speicherplatz:

BOOLEAN, INTEGER, CHAR, Aufzählungs- und Unterbereichstypen: 2 Byte; Pointer 2 Byte; REAL 5 Byte; Mengen 12 Byte.

Der dem Typ CHAR zugrunde gelegte Zeichensatz entspricht dem Commodore-Standard (ASCII erweitert).

Der Wertebereich des Typs INTEGER wird durch die vordefinierte Konstante MAXINT=32767 definiert:

```
-MAXINT-1 <= X <= MAXINT.
```

Für die Elemente von Mengen muß immer gelten:

```
0 <= ORD(X) <= 95
```

Ist aber 96 <= ORD(X) <= 255, so liefert X IN [..] den Wert FALSE. Für alle anderen Werte von ORD(X) sind Mengenoperationen undefiniert.

Wird bei der Deklaration eines Arrays oder Records das Schlüsselwort PACKED angegeben, so werden Komponenten der folgenden skalaren Typen so gepackt, daß sie nur 1 Byte Speicherplatz verbrauchen:

- Standardtypen CHAR, BOOLEAN
- Aufzählungstypen (mit weniger als 257 Elementen)
- Ausschnitt-Typen der Form A..B mit ORD(A)>=0 und ORD(B)<=255

Bemerkungen zum Packen:

Gepackte **Komponenten** der obigen Typen können nicht als Variablenparameter an Unterprogramme übergeben werden. Der Compiler erzeugt in diesem Fall die Fehlermeldung 504.

Da Packen zusätzlichen Code erfordert, ist es nur für große Files und große Tabellen im Arbeitsspeicher sinnvoll. Daher sollten bei der Übernahme von Pascal-Programmen Strings normalerweise **nicht** als PACKED ARRAY OF CHAR, sondern nur als ARRAY OF CHAR dargestellt werden.

Typverträglichkeit

Zwei Variablen haben den gleichen Typ, wenn sie entweder

- in einer Variablenvereinbarung oder
- mit dem gleichen Typbezeichner deklariert wurden.

Beispiel

```
TYPE FELD = ARRAY [0..1] OF INTEGER;
VAR  A  :ARRAY [0..1] OF INTEGER;
     B  :ARRAY [0..1] OF INTEGER;
     X,Y:ARRAY [0..1] OF INTEGER;
     L  :FELD;
     M  :FELD;
```

A und B besitzen nicht den gleichen Typ.
X und Y besitzen den gleichen Typ.
L und M besitzen den gleichen Typ.

Deshalb sind die folgenden Zuweisungen gültig:

```
X:=Y; L:=M
```

aber nicht:

```
A:=B; A:=L; M:=X; A:=X
```

Da in Pascal bei Prozedur- und Funktionsvereinbarungen sowieso Typbezeichner erwartet werden, empfiehlt sich folgende Vorgehensweise: Man vereinbart einmalig Typbezeichner (im Beispiel FELD), die man sowohl bei der Deklaration von Parametern in Unterprogrammen als auch bei der Deklaration von Variablen verwendet.

Feldbezeichner

Feldbezeichner müssen im jeweils innersten Bereich eines Records eindeutig identifizierbar sein. Es dürfen natürlich auch gleichnamige Variablen im Programm existieren.

Records mit Varianten

Die Varianten erhalten denselben Speicherplatz. Der benötigte Speicherplatz richtet sich nach der Größe der längsten Variante. Die Werte der Variantenwahl (tagfield) schränken zur Laufzeit den Zugriff auf die Varianten nicht ein. Es erfolgt also keine Prüfung.

Beispiel

```
TYPE TART = (KARTESISCH, POLAR);
     KOORDINATE =
     RECORD CASE ART: TART OF
            KARTESISCH:(X,Y:REAL);
            POLAR:     (R,PHI:REAL)
     END;
VAR A: KOORDINATE;
```

Eine Variable vom Typ Koordinate benötigt also 10 Byte Speicherplatz. Außerdem belegt die Variante X den Speicherplatz der Varianten R. Bei den Zuweisungen

```
A.ART :=KARTESISCH; A.R:= 2.0
```

erfolgt keine Fehlermeldung. Außerdem sind variante Records ohne Variantenwahl (tagfield) erlaubt.

With-Anweisung

Die Verwendung des With-Statements bringt erhebliche Verbesserungen im Laufzeitverhalten des Programms, da sämtliche Adreßberechnungen für die folgende(n) Anweisung(en) nur genau einmal ausgeführt werden.

Beispiel

```
WITH A [I,J]↑ DO  FELD5:= FELD5  - FELD3  statt
A[I,J]↑.FELD5:= A[I,J]↑.FELD5 - A[I,J]↑.FELD3
```

Sprunganweisung

Es ist nicht erlaubt, von außen in ein FOR .. DO-, CASE .. OF-, oder WITH .. DO-Statement mit GOTO zu springen, da diese zur korrekten Ausführung Zwischenergebnisse auf dem Stapel benötigen. Jedoch kann auch aus einer Prozedur oder Funktion in einen der sie umgebenden Blöcke gesprungen werden, solange die obigen Einschränkungen nicht verletzt werden.

Vorwärtsvereinbarungen

Generell müssen alle Bezeichner vor ihrem ersten angewandten Auftreten ein definierendes Auftreten besitzen. Es sind nur folgende Ausnahmen möglich:

- Die explizite FORWARD-Vereinbarung von Prozeduren und Funktionen.

- Typdeklarationen, die einen Zeiger auf einen noch nicht deklarierten Typ definieren.

Beispiele

```
PROCEDURE Q(I: INTEGER); FORWARD;
PROCEDURE P(C: CHAR);
BEGIN ... Q(4)   ... END;
PROCEDURE Q;
BEGIN ... P("A") ... END;
```

und

```
TYPE POINTER=↑NODE;
     NODE   =RECORD
               INFO:INFOTYP;
               NEXT:POINTER
             END;
```

Fallunterscheidung

Optional ist die Angabe eines ELSE-Zweiges (siehe Syntax-Diagramm). Dieser Zweig wird angesprungen, falls der Ausdruck einen Wert liefert, der durch keine Fallmarke erfaßt wird. Wird ELSE nicht angegeben, so erfolgt zur Laufzeit beim Auftreten eines Wertes ohne passende Marke die Fehlermeldung

```
"NO LABEL FOR CASE"
```

Bit-Operatoren

AND, OR und NOT können als Operanden auch INTEGER-Zahlen besitzen. Es erfolgt dann eine bitweise Verknüpfung der Operanden. Die Operandentypen BOOLEAN und INTEGER dürfen aber nicht gemischt werden. Die Operationen liefern ein Ergebnis des gleichen Typs (INTEGER bzw. BOOLEAN).

Absolut adressierte Variablen

Bei einer Variablendeklaration ist die explizite Angabe einer Adresse für die Variable möglich. Ein Anwendungsgebiet ist die gemeinsame Verwendung von Variablen aus Maschinenprogrammen und Betriebssystemadressen. Natürlich muß hierbei der Speicherplatzbedarf der Variablen (in Bytes) beachtet werden.

Beispiel

```
IRQVEKTOR = INTEGER [788];
```

Vergleiche

Die Vergleichsoperatoren (=,<>,>,<,>=,<=) können nicht nur auf alle Skalare, Reals und Strings (gleicher Länge), sondern auch auf beliebige (gepackte und ungepackte) zusammengesetzte Objekte angewendet werden, sofern beide Operanden denselben Typ haben. Der Vergleich erfolgt dann Byte für Byte (ohne Vorzeichen).

4.4.3 Reservierte Wortsymbole

and	file	not	to
array	for	of	type
begin	forward	or	until
case	function	packed	var
const	goto	procedure	while
div	if	program	with
do	in	record	
downto	label	repeat	
else	mod	set	
end	nil	then	

4.4.4 Vordefinierte Bezeichner

Die nachfolgend aufgeführten Bezeichner besitzen eine vordefinierte Bedeutung. Sie können jedoch auch im Programm mit neuer Bedeutung definiert werden.

4.4.4.1 Konstantenbezeichner

FALSE, TRUE	Konstanten vom Typ BOOLEAN (FALSE <TRUE)
MAXINT	Konstante vom Typ INTEGER (MAXINT =32767)

4.4.4.2 Typbezeichner

BOOLEAN	= (FALSE,TRUE)
INTEGER	Ganze Zahlen von -MAXINT-1 bis MAXINT
CHAR	Einzelne Zeichen
REAL	Wertebereich wie in BASIC
TEXT	= FILE OF CHAR

4.4.4.3 Variablenbezeichner

INPUT	:TEXT (Eingabedatei)
OUTPUT	:TEXT (Ausgabedatei)

4.4.4.4 Prozeduren für dynamische Objekte

Die dynamischen Objekte werden in einem getrennten Datenbereich (Heap) gespeichert. Reicht zur Laufzeit der freie Speicher nicht mehr aus, wird das Programm mit der Fehlermeldung HEAP OVERFLOW beendet.

Der Heap wird als First-in-last-out-Speicher betrieben. Daten werden an der Seite angefügt, an der sie gelöscht werden. Das Ende des Speichers wird durch einen Heappointer markiert.

Die Befehle MARK und RELEASE ersetzen die in vielen anderen Dialekten vorhandene Prozedur DISPOSE.

NEW (Zeigervariable)

Stellt Speicherplatz für eine (neue) dynamische Variable zur Verfügung. Die Zeigervariable wird mit der Adresse des neuen Objektes initialisiert.

MARK (Zeigervariable)

weist der Zeigervariablen den momentanen Wert des Heappointers zu. Ein anschließender Aufruf der Prozedur RELEASE mit dieser Zeigervariablen setzt den Heappointer auf diesen Wert zurück.

RELEASE (Zeigervariable)

Der Heappointer wird auf den Wert der Zeigervariablen gesetzt. Dieser Befehl sollte nur in Zusammenhang mit dem MARK-Befehl verwendet werden.

Beispiel

```
VAR A,B,C, HEAP: ↑INTEGER;
...
BEGIN
  ... MARK(HEAP) ...
  NEW(A); NEW(B); NEW (C)
  ... RELEASE(HEAP)
END.
```

Durch die Anweisung RELEASE(HEAP) werden alle dynamischen Variablen gelöscht, die nach der Anweisung MARK(HEAP) mit NEW erzeugt wurden. Im Beispiel sind dies A↑, B↑, und C↑.

4.4.4.5 Ein- und Ausgabe

Files werden durch Dateien unter dem Betriebssystem des C 64 realisiert. Details über die Zusammenarbeit mit dem Betriebssystem finden sich im Abschnitt 4.4.5.

OPEN (filevar, dev, sek, name)

filevar	Variable vom Typ FILE OF ...
dev	INTEGER-Ausdruck, der die Gerätenummer angibt.

sek	INTEGER-Ausdruck, der die Sekundäradresse festlegt.
name	Konstante oder Variable vom Typ ARRAY [...] OF CHAR.

Dem angegebenen File wird eine freie logische Filenummer zugewiesen und ein Aufruf der Betriebssystemroutine OPEN mit den obigen Parametern durchgeführt. EOF(filevar):=FALSE. filevar↑ ist undefiniert. Wird kein Filename benötigt, so ist auch der folgende Befehl möglich:

```
OPEN (filevar,dev,sek)
```

`REWRITE(f)`	entspricht `OPEN(f,dev,sek,name)`
`RESET(f)`	entspricht `OPEN(f,dev,sek,name);GET(F)`
`GET(f)`	entfällt für Textfiles, die mit `READ` gelesen werden.

CLOSE (filevar)

Das angegebene File wird geschlossen und die logische Filenummer wird freigegeben. Files müssen wie in BASIC geschlossen werden. Dies ist speziell vor einem erneuten OPEN erforderlich. Es können maximal 10 Files gleichzeitig geöffnet sein.

Nach CLOSE sind keine weiteren PUT-, GET-, READ- und WRITE-Operationen mit filevar zulässig. Wurde die durch CLOSE freigewordene logische Filenummer wieder für ein anderes File vergeben, können Sie beim Zugriff auf geschlossene Files nicht mit der Fehlermeldung FILE NOT OPEN rechnen.

EOF (filevar)

Das Ergebnis dieser Funktion ist vom Typ BOOLEAN. EOF(filevar) ist TRUE, falls beim Lesen der Datei filevar das Dateiende erreicht wurde. Beim Schreiben ist EOF(filevar) immer TRUE. EOF alleine ist die Abkürzung für EOF(INPUT). In dieser Dokumentation wird für jede File-Operation die Veränderung von EOF explizit beschrieben.

STATUS (filevar)

Diese nicht im REPORT aufgeführte Funktion besitzt den Ergebnistyp INTEGER. Das L-Byte enthält den Status bei der letzten E/A-Operation (READ, WRITE, GET, PUT). Dieser Wert ist wie für die BASIC-Variable

ST definiert. Einzelheiten sind wieder den Handbüchern zum C 64 zu entnehmen.

Das H-Byte enthält die logische Filenummer, unter der das Betriebssystem das File verwaltet. Der Wert ist nur zwischen OPEN und CLOSE definiert. STATUS alleine entspricht STATUS(INPUT).

Beispiel

```
IF (STATUS(DATAFILE) AND 1) THEN
   WRITELN("Zeitablauf beim Schreiben")
```

EOLN (filevar)

Standardfunktion mit dem Ergebnistyp BOOLEAN. Der Parameter muß ein Textfile sein. EOLN liefert den Wert TRUE, falls beim Lesen das Zeilenende erreicht wurde. Beim Lesen mit READ(f,...) ist EOLN(f)=TRUE, falls das letzte gelesene Zeichen ein CR (ASCII-Code 13) war. Jedoch liefert dann ein Zugriff auf die Puffervariable f↑ ein Leerzeichen (space). EOLN alleine entspricht EOLN(INPUT).

PUT (filevar)

Überträgt den Inhalt der Puffervariablen filevar↑ Byte für Byte auf das File. EOF(filevar):=TRUE.

GET (filevar)

Füllt die Puffervariable filevar↑ und setzt den Lesezeiger weiter. Ist nach dem Aufruf EOF(filevar)=TRUE, so ist der Wert von filevar↑ undefiniert.

READLN (filevar)

Überliest Zeichen in dem Textfile filevar bis zum Ende der Eingabezeile. Ist EOLN(filevar) beim Aufruf von READLN bereits TRUE, so werden keine Zeichen gelesen.

READ (filevar, Parameter, Parameter ...)

Die Ausgabe hängt vom Typ des Parameters ab:

Parameter ist eine Variable vom Typ CHAR: Es wird ein Zeichen eingelesen und der Variablen zugewiesen. Ist das Zeichen das Zeilenendezeichen, so wird ein Leerzeichen ' ' gelesen.

Parameter ist eine Variable vom Typ INTEGER oder REAL: Leerzeichen und Zeilentrennzeichen werden überlesen. Die nachfolgenden Zeichen werden bis zum nächsten Leerzeichen oder Zeilenende gelesen und als Zahl interpretiert (analog der Funktion VAL in BASIC). Ist der Parameter eine Variable vom Typ INTEGER, so wird die Zahl noch mit der Funktion INT angepaßt.

EOF:=TRUE, falls das letzte Zeichen des Files gelesen wurde.

Die Puffervariable enthält das letzte gelesene Zeichen. War das letzte gelesene Zeichen ein Zeilenendezeichen (CR), so liefert die Funktion EOLN den Wert TRUE.

WRITELN (filevar)

Schreibt ein Zeilentrennzeichen auf das angegebene File. EOF:=TRUE. EOLN:=TRUE.

WRITE (filevar, Parameter, Parameter, ...)

Parameter hat die Form:

- Ausdruck

  oder

- Ausdruck : INTEGER-Ausdruck

Die Zahl nach dem Doppelpunkt gibt die Mindestanzahl an Zeichen an, die ausgegeben werden. Falls nötig, werden dem Wert Leerstellen vorangestellt, um die angegebene Feldgröße zu erreichen. Wiederum ist die Ausgabe vom Typ der Ausdrücke abhängig:

CHAR:	Das Zeichen wird rechtsbündig im Feld ausgegeben.
INTEGER:	Die INTEGER-Zahl wird rechtsbündig ausgegeben. Vor positiven Zahlen steht ein Leerzeichen (nicht '+').
REAL:	Die Zahl wird im gleichen Format wie in BASIC ausgegeben. Eine Angabe der Anzahl der Nachkommastellen ist nicht möglich.
ARRAY[..] OF CHAR:	Der (ungepackte) String wird ausgegeben.

4.4.4.6 Arithmetische Funktionen

ORD (Ausdruck)

Diese Funktion liefert als Ergebnis eine INTEGER-Zahl, die die Position des Parameters im Wertebereich angibt (0,1,2,...). Der Ausdruck muß einen skalaren Typ besitzen (nicht REAL).

CHR (Ausdruck)

Diese Funktion liefert ein Zeichen mit dem ASCII-Code des INTEGER-Ausdruckes.

SUCC (Ausdruck)

Die Funktion liefert den Nachfolger im Wertebereich. Der Ausdruck muß einen skalaren Typ besitzen (nicht REAL). Eine Prüfung auf Bereichsüberschreitung erfolgt nur, falls bei der Übersetzung die Option (*$R+ *) des Compilers gewählt wurde.

PRED (Ausdruck)

Die Funktion liefert den Vorgänger im Wertebereich. Der Ausdruck muß einen skalaren Typ besitzen (nicht REAL). Eine Prüfung auf Bereichsüberschreitung erfolgt nur, falls bei der Übersetzung die Option (*$R+ *) des Compilers gewählt wurde.

ODD (Ausdruck)

Diese Funktion liefert den Wert TRUE, falls der Ausdruck vom Typ INTEGER ungerade ist.

ABS (Ausdruck)

Der Absolutbetrag des INTEGER- oder REAL-Ausdruckes wird berechnet. Das Ergebnis ist vom selben Typ wie das Argument.

INT (Ausdruck)

Umwandlung des reellen Argumentes in die nächstkleinere INTEGER-Zahl (wie in BASIC).

Beispiele

```
INT( 3.2) =  3
INT(-3.2) = -4
```

Also liefert INT andere Ergebnisse als die im Standard definierte Funktion TRUNC. Andererseits läßt sich ROUND(x) durch INT(x+0.5) realisieren.

SQRT(x), LN(x), EXP(x), SIN(x), COS(x), ARCTAN(x) liefern für einen INTEGER-/REAL-Ausdruck das im REPORT beschriebene Ergebnis vom Typ REAL.

TAN (x)

Liefert für einen INTEGER-/REAL-Ausdruck den Tangens vom Typ REAL. Diese Funktion ist nicht im REPORT vorgesehen.

POWER (x,y)

Liefert für zwei INTEGER-/REAL-Ausdrucke den Wert x hoch y vom Typ REAL. Diese Funktion ist ebenfalls nicht im Standard vorgesehen.

4.4.4.7 Verschiedenes

Die folgenden Systemfunktionen und -prozeduren erwarten teilweise Adressen als Parameter. Da Adressen auch Werte größer als MAXINT= 32767 annehmen, müssen größere Zahlen durch die entsprechende negative Zahl im Zweierkomplement ersetzt werden.

Beispiel

```
$FFE4 = 65508 = -28 (= 256*256 - 65508)
```

Um also die Routine GETIN=$FFE4 aufzurufen, muß man SYS(-28) schreiben.

SYS (Ausdruck)

Sprung in ein Maschinenprogramm, das an der angegebenen Stelle beginnt. Wie bei dem BASIC-Befehl werden vor und nach dem Aufruf die Prozessorregister in den folgenden Speicherzellen (dezimal) abgelegt:

```
780     Akkumulator
781     X-Register
782     Y-Register
783     Status-Register
```

POKE (Ausdruck 1, Ausdruck 2)

Schreibt den Wert von Ausdruck 2 in die Adresse Ausdruck 1 (wie BASIC).

PEEK (Ausdruck)

Diese Standardfunktion liefert den Inhalt der angegebenen Adresse.

ADDU (Ausdruck 1, Ausdruck 2)

Diese Funktion addiert die beiden INTEGER-Ausdrücke ohne Berücksichtigung des Vorzeichens. Es erfolgt keine Fehlermeldung bei Überläufen. Mit der Funktion ADDU (add unsigned) kann man also mit Adressen rechnen, ohne die Grenzen des Bereichs INTEGER (MAXINT) zu berücksichtigen.

Beispiele

```
ADDU(3,5)=8
ADDU(32767,1)=-32768
ADDU(-32768,-1)=32767
```

HALT

Beendet die Programmausführung ohne Fehlermeldung.

4.4.5 Files in Pascal 1.4

Zur Anpassung an das Betriebssystem des C 64 mußte die Behandlung von Files in einigen Details gegenüber dem REPORT geändert werden. Diese Unterschiede sind hier noch einmal zusammenfassend dargestellt:

Im Programmkopf dürfen nur die Standardfiles (INPUT/OUTPUT) angegeben werden. Externe Files werden abweichend vom REPORT nicht im Programmkopf, sondern nur in dem Block, in dem sie benötigt werden, als Variablen vom Typ FILE OF ... deklariert.

Jedes File F wird intern durch einen eigenen Deskriptor verwaltet, der neben der Puffervariablen (F↑) Informationen über EOF(F), STATUS(F) und EOF(F) enthält, die bei jeder E/A-Operation mit diesem File aktualisiert werden.

Files können natürlich Bestandteil anderer Datenstrukturen (Array, Record, aber nicht File) sein und auch in Prozeduren oder Funktionen **rekursiv** definiert werden, solange die Parameter der OPEN-Befehle geeignet gewählt werden (siehe Abschnitt 4.4.5.1).

Statt der Standardprozeduren RESET/REWRITE müssen OPEN- und CLOSE-Prozeduren wie in BASIC aufgerufen werden:

`REWRITE(F)` entspricht `OPEN(f,dev,sek,name)`
`RESET(f)` entspricht `OPEN(f,dev,sek,name);GET(F)`

GET(f) entfällt bei RESET für Textfiles, die mit READ gelesen werden.

Hier soll noch einmal daran erinnert werden, daß GET und PUT Files erzeugen, die die interne Repräsentation der Daten benutzen, so daß diese Files (insbesondere vom Typ FILE OF CHAR) nicht direkt ausgedruckt oder von BASIC-Programmen verwendet werden können. Falls ein Ausdruck oder eine Weiterverarbeitung erwünscht ist, sollten die Prozeduren READ(LN)/WRITE(LN) verwendet werden. Vorteile von GET und PUT sind die Klarheit des Konzeptes, die kompakte Speicherung der Daten und die schnellere Ein- und Ausgabe.

Abweichend vom REPORT enthält die Puffervariable beim Lesen von Textfiles nicht das nächste, sondern das zuletzt gelesene Zeichen. Also besteht keine Möglichkeit, über die Puffervariable ein Zeichen *vorauszuschauen.*

Zur Bearbeitung von Textfiles dienen die Prozeduren READ(LN) und WRITE(LN), die durch die Änderung der Bedeutung der Puffervariablen auch für Dialogprogramme geeignet sind.

Ausdrücklich sei darauf hingewiesen, daß STATUS(f) nur durch File-Operationen auf dem File f beeinflußt wird, so daß STATUS(F) (anders als in BASIC die Variable ST) nicht durch Ein- oder Ausgabe auf einem anderen File als f beeinflußt werden kann.

Schließlich ist es möglich, die Standardeingabe- und -ausgabefiles anderen Geräten als der Tastatur und dem Bildschirm zuzuordnen:

Beispiele

`OPEN(OUTPUT,4,0)` Ausgaben auf den Drucker
`OPEN(INPUT,8,3,name)` Eingaben vom File 'name'

Diese Umleitung der Ausgabe und Eingabe kann durch die Befehle CLOSE(OUTPUT) bzw. CLOSE(INPUT) wieder rückgängig gemacht werden. Diese Optionen sind besonders zum Protokollieren von Bildschirmausgaben und zum Einlesen von Kommandosequenzen von einer Datei sinnvoll.

4.4.5.1 Übernahme von Programmen mit Files

Wie im REPORT definiert, können die Prozeduren READ und WRITE nur auf Files vom Typ FILE OF CHAR angewendet werden. Soll ein PASCAL-Programm umgeschrieben werden, das diese Prozeduren auch für Files mit anderem Grundtyp verwendet, so können diese Befehle wie folgt ersetzt werden:

`READ (F,X)` wird ersetzt durch `X:=F↑; GET(F)`
`WRITE(F,X)` wird ersetzt durch `F↑:=X; PUT(F)`

Außerdem muß darauf geachtet werden, daß eine Prozedur, die sich rekursiv aufruft, ein lokales File mit korrekten Parametern eröffnet. Um eine Anpassung möglichst einfach vorzunehmen, kann man das Include-File FILE.INC benutzen. Es definiert die Prozeduren RESET und REWRITE. Genaue Hinweise zur Benutzung und ein Beispielprogramm sind in Abschnitt 2.16 gegeben.

Die Definitionen in FILE.INC werden an den Anfang des Hauptprogrammes gesetzt. Jedes lokale File wird am Beginn des Blockes seiner Deklaration mit ALLOC definiert. Außerdem wird im selben Block als letzte Anweisung die FREE-Prozedur für alle lokalen Files aufgerufen. Zwischen ALLOC und FREE darf kein File mit CLOSE geschlossen werden.

Diese Routinen beruhen auf dem folgenden Prinzip: Durch ALLOC weist das Laufzeitsystem jedem File eine eindeutige logische Filenummer zu, die bei RESET/REWRITE zur Identifikation des Files auf der Diskette verwendet wird. Das File KOMMANDO wird zum Löschen von alten Dateien bei REWRITE benutzt. Die logische Filenummer wird sowohl als eindeutige Sekundäradresse für die Floppy als auch als Teil des Dateinamens auf der Diskette verwendet.

4.4.6 Aktive Kommentare

Kommentare können beliebig im Quelltext eingefügt werden. Sie sollten jedoch nicht mit einem Dollarzeichen '$' beginnen, da dieses sogenannte **aktive Kommentare** einleitet.

4.4.6.1 Bereichstests

Der Compiler besitzt einen *Schalter*, mit dem die Erzeugung von Code für Laufzeittests gesteuert werden kann. Beim Beginn der Übersetzung ist der Schalter auf *aus* gestellt. Nach einem Kommentar der Form

```
(*$R+     *)
```

steht der Schalter für Bereichstests (Range Check) auf *ein*. Der Compiler erzeugt dann Code, um bei der Laufzeit des Programmes die folgenden Operationen zu prüfen:

Zuweisungen von Werten eines Grundtyps an Variablen eines Unterbereichs. Dies betrifft auch die Grenzen von FOR .. DO-Schleifen, bei denen die Laufvariable einen Unterbereichstyp besitzt.

Mengenoperationen der Form [A] [A..B] und B IN X auf gültige Werte von A und B (d.h. 0<=ORD(A),ORD(B)<=95). Indizierungen von Arrays der Form ARRAY [A..B] OF ...

Ergebnisse der Standardprozeduren SUCC(X) / PRED(X), bei denen X ein Ausdruck eines Aufzählungs- oder Unterbereichstyps ist.

Tritt ein ungültiger Wert auf, so bricht die Programmausführung mit der Fehlermeldung 'VALUE OUT OF BOUNDS' ab. Außerdem werden nacheinander die Ordinalwerte des fehlerhaften Wertes, der unteren und der oberen Bereichsgrenze angegeben. Ein Beispiel für die Interpretation der Fehlermeldung ist in Abschnitt 2.12 gegeben.

Der Schalter wird mit dem folgenden aktiven Kommentar auf *aus* gestellt:

```
(*$R-     *)
```

Diese Kommentare sollten nicht direkt neben Ausdrücken stehen, sondern durch mindestens ein Symbol von dem Ausdruck getrennt sein. Sonst könnte es passieren, daß der Schalter *zu früh* ein- oder ausgeschaltet wird.

4.4.6.2 Include-Files

Wie bereits in vorangegangenen Abschnitten erwähnt, kann der Compiler Teile des Quelltextes bei der Übersetzung von Diskette lesen.

Da man nur 8 Kbyte Quelltext mit dem Editor im Pascal-System auf einmal bearbeiten kann, geht man bei großen Programmen am besten wie folgt vor:

Man entwickelt zunächst einzelne Teile des Programmes und testet diese. Solche *Module* werden dann mit dem Editor auf Diskette gespeichert. Um nun den Text eines solchen Moduls im Programm einzufügen, genügt es, an der entsprechenden Position im Quelltext einen aktiven Kommentar einzufügen:

```
(*$"filename" *)
```

Erreicht der Compiler bei der Übersetzung diesen Kommentar, so wird der Rest der Zeile ignoriert und ab dieser Stelle der Programmtext von dem Text "filename" auf der eingelegten Diskette gelesen. Am Dateiende setzt der Compiler die Übersetzung aus dem Speicher fort.

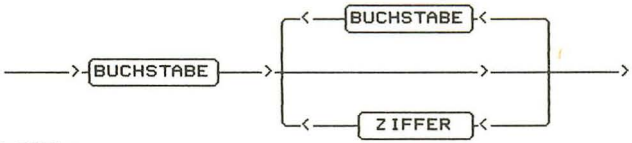
Die Include-Files selbst sind normale Texte, die mit dem Editor mit SAVE oder END auf der Diskette gespeichert wurden.

Es sind beliebig viele Include-Files in einem Programm möglich. Außerdem kann auch in einem Include-File ein anderes Include-File aufgerufen werden. Jedoch sind **keine** Schachtelungen möglich, da der Compiler am Ende eines Files immer zur Übersetzung aus dem Speicher zurückkehrt.

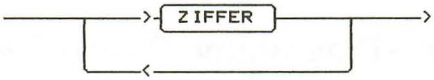
Anhang A: Syntax-Diagramme Pascal 1.4

SYNTAXDIAGRAMME PASCAL 1.4

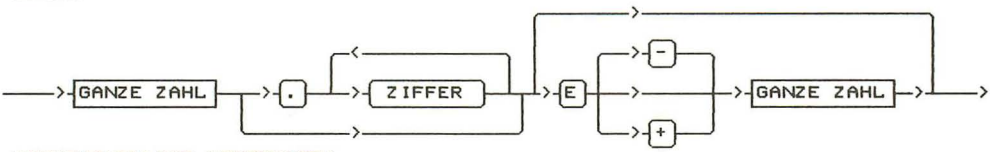
BEZEICHNER:



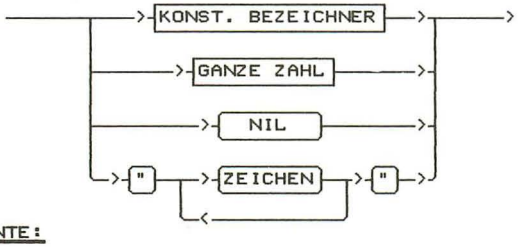
GANZE ZAHL:



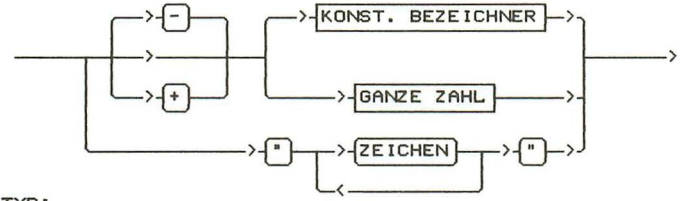
ZAHL:



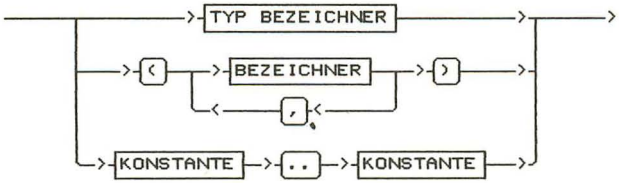
VORZEICHENLOSE KONSTANTE:



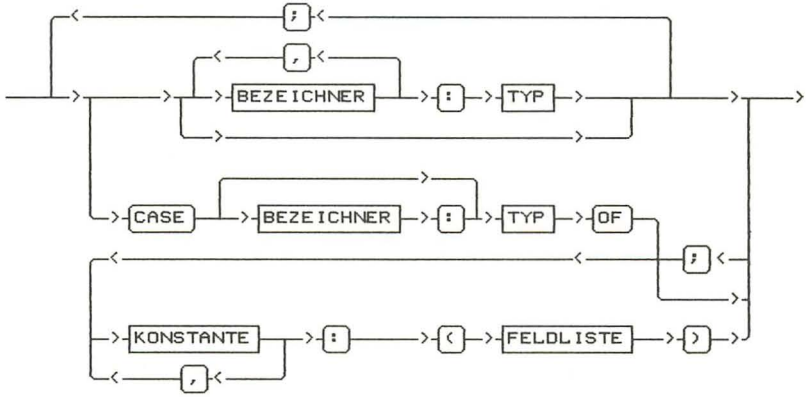
KONSTANTE:



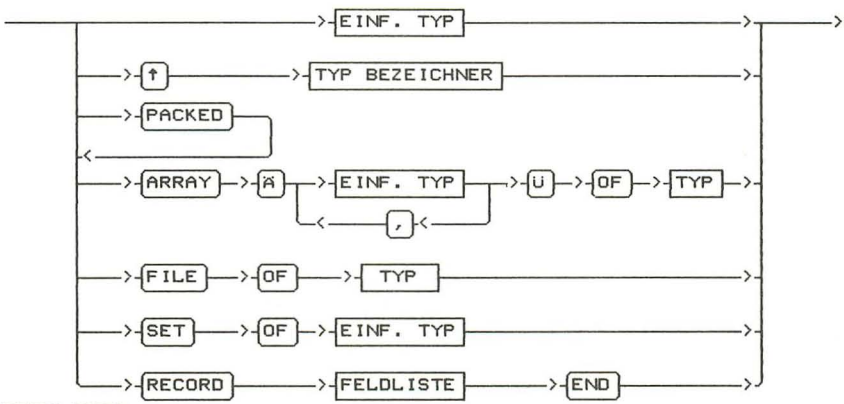
EINF. TYP:



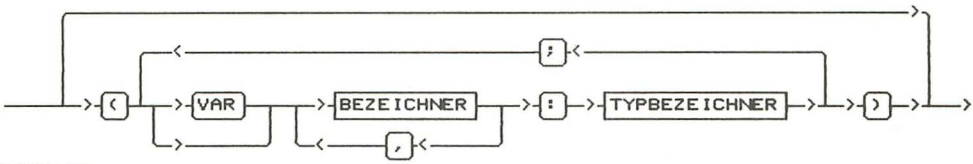
FELDLISTE:



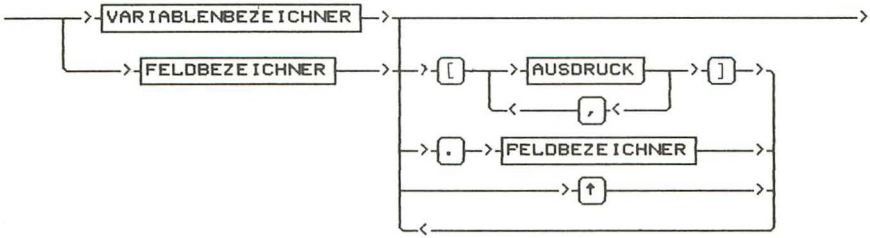
TYP:



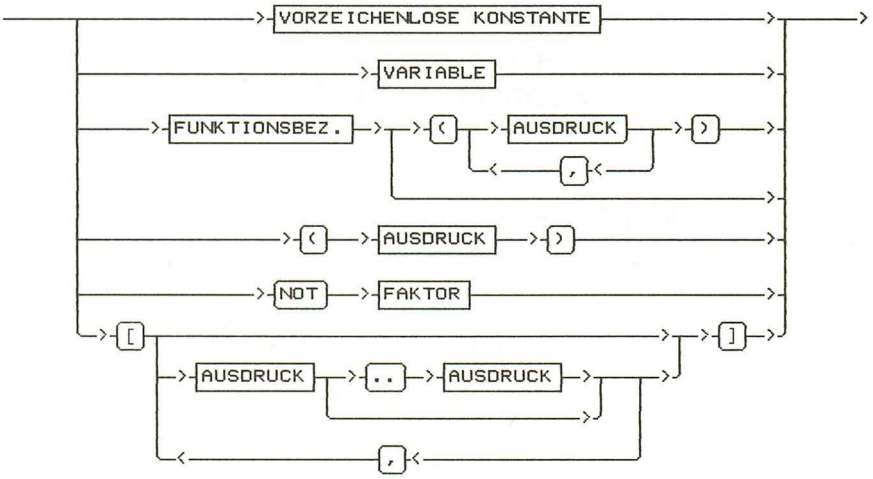
PARAMETERLISTE:



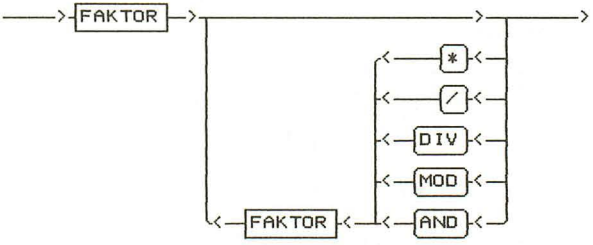
VARIABLE:



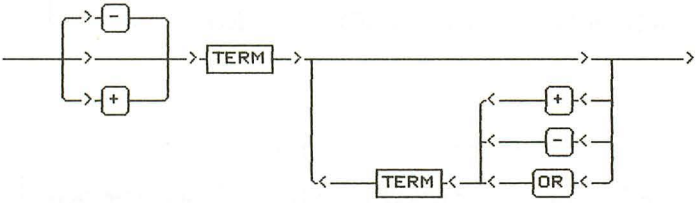
FAKTOR:



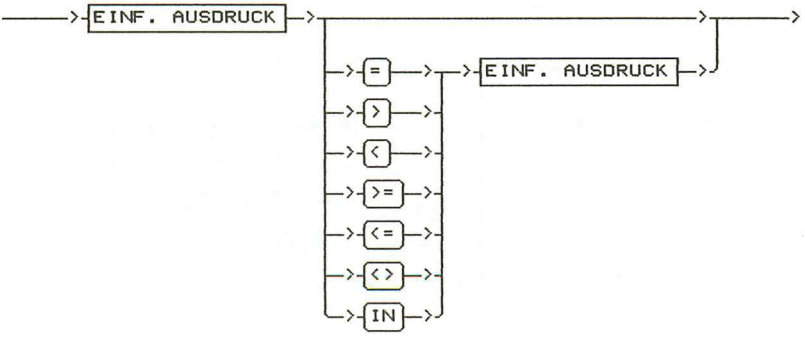
TERM:



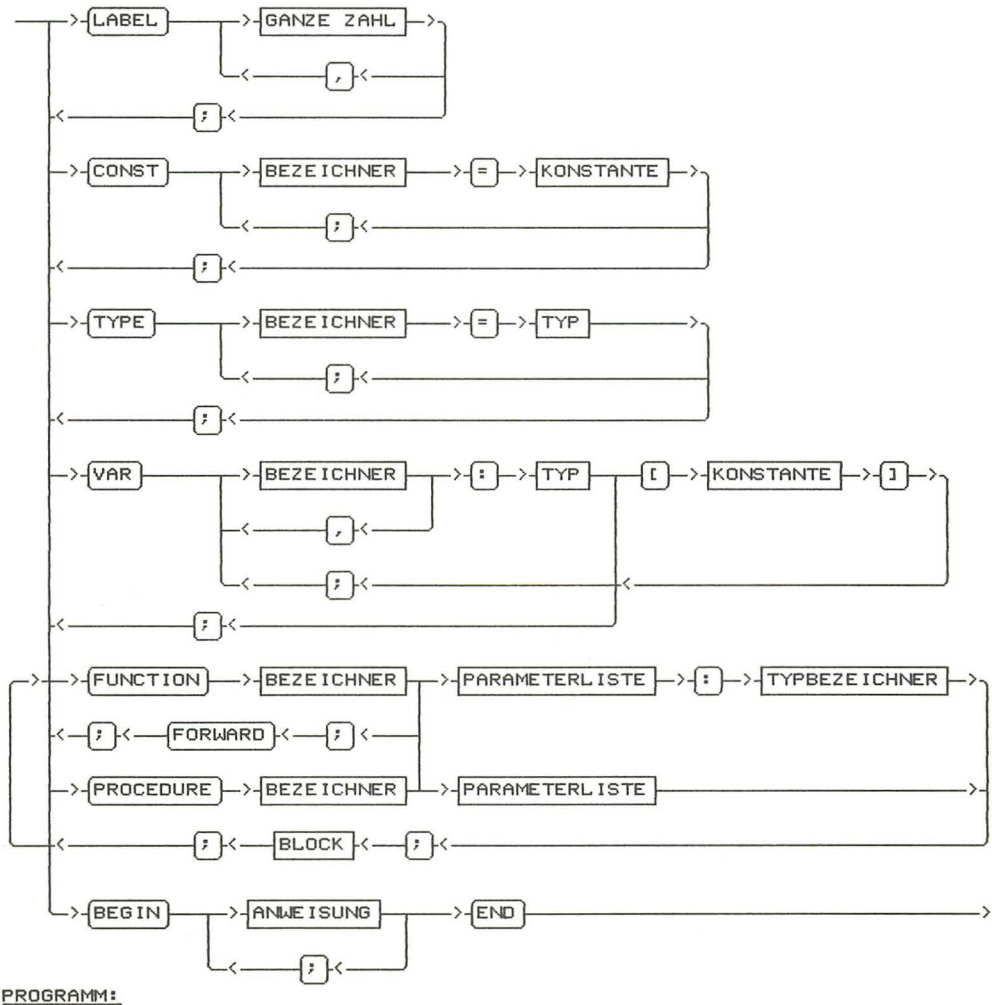
EINF. AUSDRUCK:



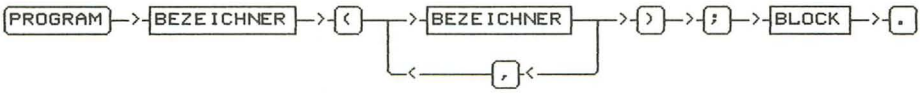
AUSDRUCK:



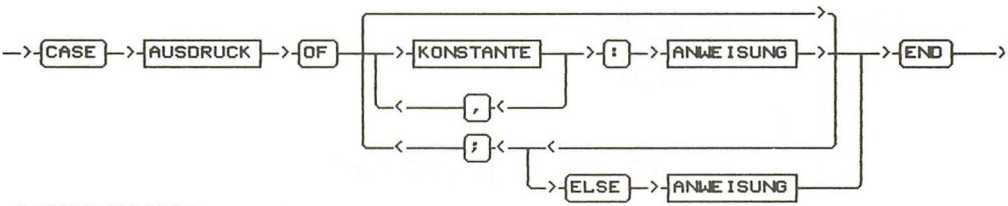
BLOCK:



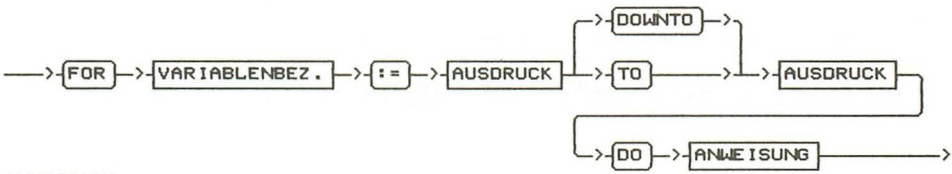
PROGRAMM:



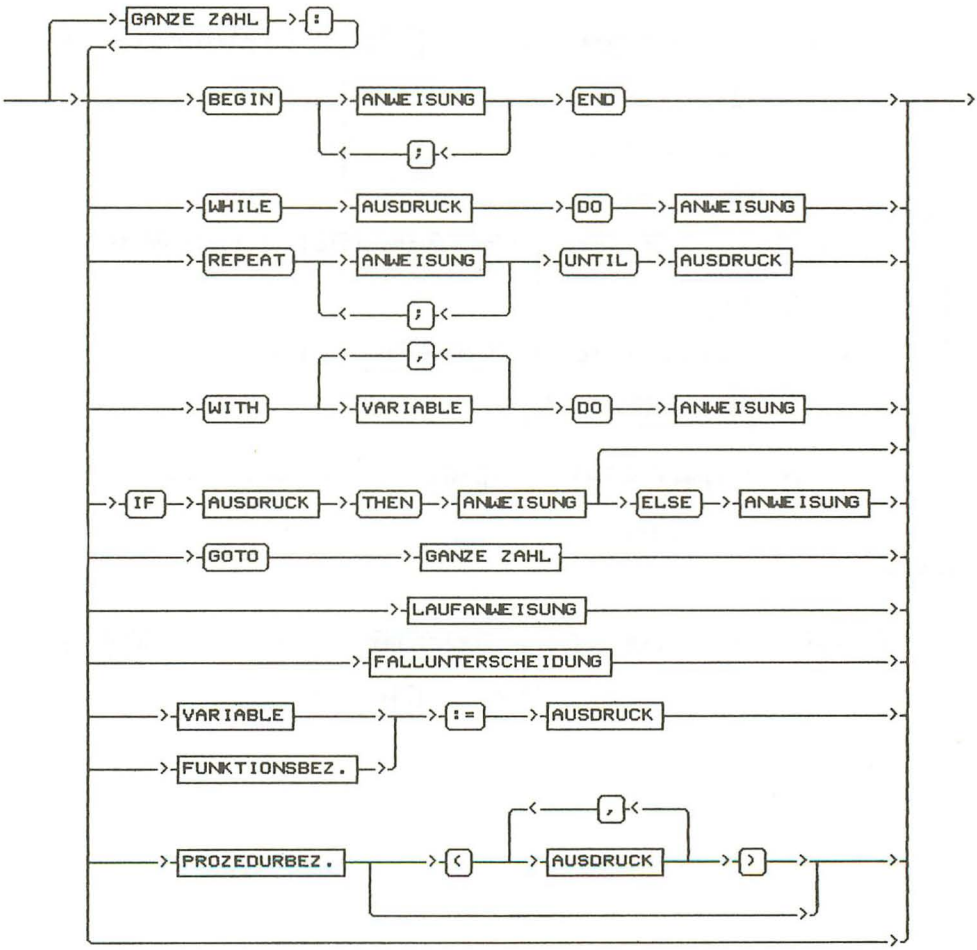
FALLUNTERSCHEIDUNG:



LAUFANWEISUNG:



ANWEISUNG:



Anhang B: Fehlernummern Pascal 1.4

0	Position eines Laufzeitfehlers (Option LOCATE ADDRESS)
1	Fehler in Typangabe
2	Bezeichner erwartet
3	'PROGRAM' erwartet
4	')' erwartet
5	':' erwartet
8	'OF' erwartet
9	'(' erwartet
11	'[' erwartet
12	']' erwartet
13	'END' erwartet
14	';' erwartet
15	INTEGER erwartet
16	'=' erwartet
17	'BEGIN' erwartet
20	'.' erwartet
22	',' erwartet
50	Fehler in Konstante
51	':=' erwartet
52	'THEN' erwartet
53	'UNTIL' erwartet
54	'DO' erwartet
55	'TO'/'DOWNTO' erwartet
59	Fehler in Variable (Variablenbezeichner erwartet)
60	String ist hier nicht zulässig
101	Bezeichner zweimal deklariert
102	Untere Grenze übersteigt obere Grenze

103 Bezeichner ist nicht von der richtigen Klasse
104 Bezeichner nicht deklariert
105 Vorzeichen hier nicht zulässig
106 Zahl erwartet
107 Inkompatible Unterbereichstypen
109 Grundtyp muß Skalartyp oder Unterbereich sein (nicht REAL)
110 Typ des Tagfields muß Skalartyp oder Unterbereich sein
111 Konstante nicht kompatibel mit dem Tagfield
113 Indextyp muß Skalartyp oder Unterbereich sein (nicht REAL oder INTEGER)
116 Falscher Typ eines Parameters für Standardprozedur
117 Ungelöste Vorwärtsvereinbarung (Typbezeichner oder Prozedur)
118 Undeklarierter Typbezeichner in Variablendeklaration
120 Ergebnistyp einer Funktion muß Skalartyp, Unterbereich oder Zeiger sein
121 File als Wertparameter nicht zulässig
123 Ergebnistyp fehlt im Funktionskopf
125 Falscher Typ eines Parameters für Standardfunktion
126 Anzahl der Parameter stimmt nicht mit Deklaration überein
127 Unzulässige Parameter-Substitution
129 Operandentypen nicht kompatibel
130 Ausdruck ist nicht vom Typ Menge
131 Nur Test auf Gleichheit zulässig
132 Test auf echtes Enthaltensein nicht zulässig
134 Unzulässiger Operandentyp
135 Inkompatible Strings (Länge stimmt nicht überein)
136 Elementtyp einer Menge muß Skalartyp oder Unterbereich sein
137 Elementtypen nicht kompatibel
138 Variable ist nicht vom Typ Array
139 Indextyp entspricht nicht der Deklaration
140 Variable ist nicht vom Typ Record
141 Variable ist weder Zeiger noch File
143 Laufvariable besitzt einen unzulässigen Typ
144 Ausdruck hat einen unzulässigen Typ
145 Typkonflikt
146 Zuweisung von Files nicht zulässig
147 Typ der Fallmarke nicht kompatibel mit CASE-Ausdruck
152 Feld existiert in diesem Record nicht
154 Aktueller Parameter muß eine Variable sein
155 Laufvariable muß eine lokale, nicht gepackte Variable sein
159 REAL oder String nicht als Tagfield zulässig
161 FORWARD hier nicht zulässig
165 Label mehrfach (im Anweisungsteil) definiert

166 Label mehrfach (im Vereinbarungsteil) deklariert
167 Label nicht deklariert
168 Undefiniertes Label im vorherigen Block
171 Variable muß vom Typ File sein
172 Fehlende Parameter für Standardprozedur
173 File ist nicht vom Typ TEXT (PUT oder GET benutzen!)
174 Standardfile wiederdefiniert
400 Zu viele Fallmarken in Case-Anweisung
401 Zu viele Labels im Programm
402 Zu viele Bezeichner im Programm

500 Operandentypen müssen INTEGER sein
501 Ordnungszahlen des Grundtyps nicht im Bereich 0..95
502 Typ BOOLEAN oder INTEGER erwartet
503 Externe Files werden hier nicht angegeben
504 Variablenparameter darf nicht gepackt sein
505 Standardfile OUTPUT muß deklariert werden
506 'NIL' ist hier nicht zulässig
510 FORWARD-Deklaration muß in derselben Schachtelungstiefe erfolgen
511 Ganze Zahl erwartet
512 Parameter dürfen keine absolute Adresse erhalten

Anhang C: Laufzeitfehler

STACK OVERFLOW	Am Prozeduranfang: kein Speicherplatz für die lokalen Variablen. Sonst: kein Platz für Zwischenergebnisse.
INTEGER OVERFLOW	Bereichsüberschreitung bei ganzen Zahlen.
DIVISION BY 0	Division durch Null bei MOD oder DIV.
NO LABEL IN CASE	Keine Fallmarke für diesen Wert in der Case-Anweisung gefunden.
HEAP OVERFLOW	Bei NEW ist auf dem Heap kein Platz für eine neue dynamische Variable vorhanden.
VALUE OUT OF BOUNDS	In einem Ausdruck tritt ein illegaler Wert auf (siehe Abschnitt 4.4.6.1). Es werden folgende Ordinalwerte ausgegeben: ORD (fehlerhafter Wert) ORD (untere Bereichsgrenze) ORD (obere Bereichsgrenze).
BREAK	Programm wurde mit RUN/STOP & RESTORE unterbrochen.

TOO MANY FILES OPEN	Es dürfen maximal 10 Files gleichzeitig geöffnet sein.
FILE NOT FOUND	Bei OPEN konnte das angegebene File nicht gefunden werden (siehe Handbücher).
DEVICE NOT PRESENT	Bei READ, WRITE, GET oder PUT wurde festgestellt, daß das Peripheriegerät nicht aktiv ist.
NOT INPUT FILE	Dieses Gerät (z.B. der Bildschirm) kann keine Daten liefern.
NOT OUTPUT FILE	An dieses Gerät (z.B. die Tastatur) kann man keine Daten senden.
MISSING FILE NAME	Bei OPEN muß bei diesem Gerät ein Filename angegeben werden.
ILLEGAL DEVICE NUMBER	Diese Geräteadresse (bei OPEN) ist nicht zulässig.
ILLEGAL QUANTITY	Beim Aufruf einer Standardfunktion, die reelle Argumente besitzt, wurden illegale Argumente übergeben. Diese Fehlermeldung tritt auch bei der Funktion INT auf.
OVERFLOW	Bei einer Operation mit reellen Zahlen trat eine Bereichsüberschreitung auf.
DIVISION BY ZERO	Bei der Division mit (/) ist der zweite Operand 0.0.

Anhang D: Operatoren in Pascal

Operator	Operation	Operanden-typen	Ergebnistyp
+ (Vorz.)	Identität	INTEGER, REAL	wie Operand
- (Vorz.)	Vorzeichen-umkehr	INTEGER, REAL	wie Operand
+	Addition	INTEGER, REAL	INTEGER, REAL
	Vereinigungs-menge	Menge	Menge
-	Subtraktion	INTEGER, REAL	INTEGER, REAL
	Differenz-menge	Menge	Menge
*	Multiplikation	INTEGER, REAL	INTEGER, REAL
	Schnittmenge	Menge	Menge
DIV	Division mit Rest	beide INTEGER	INTEGER

MOD	Divisionsrest	beide INTEGER	INTEGER
/	Division	INTEGER, REAL	INTEGER, REAL
=	gleich	Skalar, Pointer Menge, String	BOOLEAN
< >	ungleich	Skalar, Pointer Menge, String	BOOLEAN
<	kleiner	Skalar, String	BOOLEAN
>	größer	Skalar, String	BOOLEAN
< =	kleiner oder gleich	Skalar, String	BOOLEAN
	Test auf Teilmenge	Menge	
> =	größer oder gleich	Skalar, String	BOOLEAN
	Test auf Obermenge	Menge	
IN	Test auf Zugehörigkeit zur Menge	1. Operand Skalar 2. Operand Menge	BOOLEAN
NOT	nicht	BOOLEAN (INTEGER)	BOOLEAN (INTEGER)
OR	oder	BOOLEAN (INTEGER)	BOOLEAN (INTEGER)
AND	und	BOOLEAN (INTEGER)	BOOLEAN (INTEGER)

Bemerkung

Die Verwendung von INTEGER-Operanden bei den logischen Operatoren NOT, AND und OR ist nur in Pascal 1.4 erlaubt.

Bei den relationalen Operatoren außer IN sind in Pascal 1.4 auch beliebige zusammengesetzte Typen (RECORD, ARRAY) erlaubt. Der Vergleich erfolgt byteweise (ohne Vorzeichen).

Anhang E: Literaturhinweise

1. Jensen, K., Wirth, N.: PASCAL, User Manual and Report Lecture Notes in Computer Science, Vol. 18. Springer-Verlag 1974.

2. Wirth, N.: Algorithmen und Datenstrukturen. LAMM Teubner-Studienbücher Informatik 1979.

3. Barron, D. W. (Editor): PASCAL - The Language and its Implementation. Wiley Series in Computing, John-Wiley and Sons 1981.

4. Harrington, S.: Computer Graphics (A Programming Approach). International Student Edition, Mc Graw-Hill 1983.

5. Wirth, N.: Systematisches Programmieren. Eine Einführung.

6. Knuth, D. E.: The Art of Computer Programming. Band 1 und Band 3. Addison Wesley 1973.

7. Maurer, H.: Datenstrukturen und Programmierverfahren. LAMM Teubner-Studienbücher Informatik 1974.

8. Mehlhorn, K.: Effiziente Algorithmen. LAMM Teubner-Studienbücher Informatik 1977.

Anhang F: Index

ABS 36f., 188
ADDU 190
Adresse 107
ALLOC 109, 192
ALT 131
AND 29, 42
Anonym 122
Anweisung 28, 45ff.
  bedingte 47ff.
  Leer- 47
ARCTAN 37
Array 59ff.
  eindimensionaler 60
  mehrdimensionaler 66
Ausdrücke 29

Baum 132
BEGIN 26, 28, 46, 49
Bezeichner 22, 75, 178, 183
Bezeichner, Sichtbarkeit 71
Bit-Operatoren 182
Blockstruktur 45
Block-Zuweisungen 68
BND 155
BOOLEAN 35, 42, 178, 183
Bottom up 106

Case-Anweisung 50, 104
CASE-OF 181
CHANGE 41, 148, 157, 159, 163
CHAR 35, 40, 118, 178
CHR 41, 188
CLOSE 116, 139, 185, 192
COL 155
Compiler 12, 17, 172
CONST 45
COPY 168
COS 30, 37
Cursorsteuerung 156

Datei 107
Datentypen, elementare 35
Deklaration 27, 44, 71, 74
Dezimale Adresse 18
DISPOSE 131, 183, 187
DIV 29, 36
DOWNTO 56
Dynamisch 121

Editor 13, 15, 148, 153 ff., 170f.
ELSE 47ff., 181
END 26, 28, 46, 49
EOF 108, 185
EOLN 117, 186
EOLN(F) 118
Ergebnistyp 81
EXP 30, 37

FALSE 42, 183
File 107, 139, 190
FIND 157, 159, 162
For-Anweisung 55
FOR...DO 181
FREE 109

Ganze Zahl 23
GET 108, 186
GETCH 148
GOTO 57

HALT 190
HEAP-OVERFLOW 183

If-Anweisung 48
IN 29, 96
Index 60f.
Inkarnation 83
INPUT 26, 115, 117, 165
INTEGER 35f., 39f., 178, 183
Interpreter 13
ITEM 113

KEY 113
Kommentar, aktiver 25, 193
Konstanten 44

Label 57
Lauf 113
Laufzeitsystem 17
Line-Command 16, 159
Liste 121, 125
Listenstrukturen 126
LN 37
Lokalität 75

MARK 131f., 184
Marke 64
Matrix 66
MAXINT 36, 178
Menge 96
MERGE 113
MOD 29, 36
MSk 155

NATURALMERGE 113
NEW 122, 130, 184
NIL 123
NOT 42

Objekt-Programm 13
ODD 42, 188
OPEN 116, 139, 184
Operanden 29
Operatoren 29
OR 29, 42
ORD 41, 188
OUT 148
Output 26, 115, 166

Parameter 78
  aktuelle 79
  formale 79
  Funktions- 81
  Variablen- 79
PEEK 144, 190
POKE 144, 190
POWER 189
Pre check loops 53
PRED 92, 188
Primary-Command 16, 154, 158
Primitiven 102
PROCEDURE 74
PROF 155
Prozeduraufruf 73
Puffervariablen 107
PUT 108, 186

Quelltext 13
Quicksort 88, 90

READ 32, 117, 119, 186
READLN 119, 186
REAL 35, 39, 183
REAL-Zahlen 39

Record-Typen 99ff., 180
Rekursion 82
Rekursiver Algorithmus 86
REL 144
Relative Dateien 144
RELEASE 131, 183f.
Repeat-Anweisung 54
REPORT 177, 191
RESET 108f., 140
REWRITE 108f., 140
ROUND 40

Semikolon 47
Sequentiell 108
SIN 30, 37
Sonderzeichen 24
Sortieralgorithmus 87
Sortieren 63
Sprunganweisung 57, 181
SQR 37f., 40
SQRT 37
STATUS 185, 191
String 65
SUCC 92, 188
Symbole 21
Syntax 21
Syntax-Diagramme 22, 195ff.
SYS 146, 189

TAB 155
TAPE 109, 113
Textfenster 153, 156
TO 56
TOP 156, 159
TRUE 42
TRUNC 40
Typ 28, 35, 71, 92ff., 178f.
Typbezeichner 103, 106f.
Typdeklaration 92

VAR 28
Variablen 27, 29, 182
Vergleiche 182

Wert 79
While 51ff.
WITH-DO 100, 180f.
Wortsymbole 24, 182
WRITE 30, 115ff., 187
WRITELN 30, 187

Zahlen 22
Zeichen 40
Zeichenfolgen 115
Zeiger 121
Zeigertypen 121, 124
Zeigervariable 122, 184
Zuweisung 28

Weitere Fachbücher aus unserem Verlagsprogramm

COMMODORE 16/116

W. Besenthal/J. Muus

Alles über den C16

Juli 1986, 292 Seiten

Dieses Buch ist ein Lern- und Nachschlagewerk für jeden Commodore-Anwender. Es ist übersichtlich gegliedert und enthält alle Informationen, die für die praktische Arbeit am Computer notwendig sind: BASIC-Kurs mit Beispielen, Strukturiertes Programmieren, Dateiverwaltung, Grafikprogrammierung, Tips & Tricks.

Best.-Nr. MT 90385, ISBN 3-89090-385-1
(sFr. 35,90/öS 304,20) **DM 39,-**

COMMODORE 64

F. Ende

Das große Spielebuch - Commodore 64

1984, 141 Seiten

46 Spielprogramme · Wissenswertes über Programmiertechnik · praxisnahe Hinweise zur Grafikherstellung · alles über Joystick- und Paddleansteuerung · das Spielebuch mit Lerneffekt.

Best.-Nr. MT 603, ISBN 3-922120-63-6
(sFr. 27,50/öS 232,40) **DM 29,80**
Best.-Nr. MT 604 (Beispiele auf Diskette)
(sFr. 38,-/öS 342,-) **DM 38,-***

* inkl. MwSt. Unverbindliche Preisempfehlung.

S. Krute

Grafik & Musik auf dem Commodore 64

1984, 336 Seiten

68 gut strukturierte und kommentierte Beispielprogramme zur Erzeugung von Sprites und Klangeffekten · Sprite-Tricks · Zeichengrafik · hochauflösende Grafik · Musik nach Noten · spezielle Klangeffekte · Ton und Grafik · für fortgeschrittene Anfänger, die alle Möglichkeiten des C64 ausnutzen wollen.

Best.-Nr. MT 743, ISBN 3-89090-033-X
(sFr. 35,-/öS 296,40) **DM 38,-**

H. L. Schneider/W. Eberl

Das C64-Profihandbuch

1985, 413 Seiten

Ein Buch, das alle wichtigen Informationen für professionelle Anwendungen mit dem C64 enthält. Mit allgemeinen Algorithmen, die auch auf andere Rechner übertragbar sind, und vielen Utilities, getrennt nach BASIC- und Maschinenprogrammen. Besonders nützlich: erweiterte PEEK- und POKE-Funktionen.

Best.-Nr. MT 749, ISBN 3-89090-110-7
(sFr. 47,80/öS 405,60) **DM 52,-**

W.-J. Becker/M. Folprecht

Programmieren unter CP/M mit dem C64

1985, 290 Seiten

Wenn Sie wissen wollen, wie das Betriebssystem CP/M-2.2 auf dem C64 implementiert ist, außerdem einiges über Turbo-Pascal, Nevada-Fortran, MBASIC-80 erfahren wollen, dann ist dieses Buch genau richtig für Sie! Mit Schaltplänen zur eigenen Fertigung des CP/M-Moduls. Für eingefleischte C64-Profis.

Best.-Nr. MT 751, ISBN 3-89090-091-7
(sFr. 47,80/öS 405,60) **DM 52,-**

J. Mihalik

35 ausgesuchte Spiele für Ihren Commodore 64

1984, 141 Seiten

Programmieren Sie selbst 35 faszinierende Spiele · geschrieben in Commodore-64-BASIC · mit Farbe, Grafiken und Ton · Vorschläge zur Programmabwandlung · für kreative Computerfans, die ihre Programmierkenntnisse vertiefen wollen!

Best.-Nr. MT 774, ISBN 3-89090-064-X
(sFr. 23,-/öS 193,40) **DM 24,80**

W. Kassera/F. Kassera

C64 - Programmieren in Maschinensprache

1985, 327 Seiten inklusive Beispieldiskette

In diesem Buch finden Sie über 100 Beispiele zur Assembler-Programmierung mit viel Kommentar und Hintergrundinformationen: Das Schreiben von Maschinenprogrammen · Rechnen und Texten mit vorhandenen Routinen · Bedienung von Drucker und Floppy · Wie man BASIC- und Maschinenprogramme verknüpft · Erstellen von eigenen Befehlen in Modulform. Für Profis!

Best.-Nr. MT 830, ISBN 3-89090-168-9
(sFr. 47,80/öS 405,60) **DM 52,-**

P. W. Dennis/G. Minter

Spiele für den Commodore 64

1984, 196 Seiten

Bewährte alte und raffinierte neue Spiele für Ihren Commodore 64 · klar und übersichtlich gegliederte Programme im Commodore-BASIC · Sie lernen: wie man Unterprogramme einsetzt · eine Tabelle aufbauen und verarbeiten · Programme testen · mit vielen Programmiertricks · für Anfänger.

Best.-Nr. MT 90074, ISBN 3-89090-074-7
(sFr. 23,-/öS 193,40) **DM 24,80**
Best.-Nr. MT 795 (Beispiele auf Diskette)
(sFr. 38,-/öS 342,-) **DM 38,-***

* inkl. MwSt. Unverbindliche Preisempfehlung.

K. Schramm

Die Floppy 1541

1985, 434 Seiten

Für alle Programmierer, die mehr über ihre VC-1541-Floppystation erfahren wollen. Der Vorgang des Formatierens · das Schreiben von Files auf Diskette · die Funktionsweise von schnellen Kopier- und Ladeprogrammen · viele fertige Programme · Lesen und Beschreiben von defekten Disketten · Für Einsteiger und für fortgeschrittene Maschinensprache-Programmierer.

Best.-Nr. MT 90098, ISBN 3-89090-098-4
(sFr. 45,10/öS 382,20) **DM 49,-**
Best.-Nr. MT 710 (Beispiele auf Diskette)
(sFr. 29,90/öS 269,10) **DM 29,90***

* inkl. MwSt. Unverbindliche Preisempfehlung.

Die angegebenen Preise sind Ladenpreise

607326/1

Sie erhalten Markt & Technik-Bücher bei Ihrem Buchhändler

Markt & Technik Verlag AG Unternehmensbereich Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München

FLORIAN MATTHES,
geboren 1963 in Frankfurt am Main. Abitur 1981. Ab 1983 Studium der Informatik mit Nebenfach Physik an der Universität Frankfurt. Seit 1985 wissenschaftlicher Mitarbeiter am Lehrstuhl für Technische Informatik. Einige Veröffentlichungen in Mikrocomputer-Fachzeitschriften und freiberufliche Arbeit in der DV-Branche.

Pascal mit dem C64

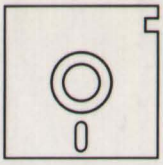
Dem Buch liegt ein leistungsfähiges Pascal-System mit Beispiel-Programmen auf Diskette bei.
Buch und Compiler ermöglichen jedem Besitzer eines C64 den Einstieg in die moderne Programmiersprache Pascal.
Dem Anfänger wird ein Einführungskurs in Pascal geboten, wobei viele überschaubare Beispiele aus der Praxis und Übungsaufgaben zum aktiven Lernen mit dem C 64 auffordern. Beim Programmieren wird er durch eine ausführliche Bedienungsanleitung des Systems unterstützt.
Für den Pascal-Profi gibt es neben nützlichen Beispielprogrammen ein spezielles Kapitel mit Tips und Tricks.
Der Compiler akzeptiert den gesamten Sprachumfang mit einigen Erweiterungen. Der Compiler bildet mit seinem sehr komfortablen Full-Screen-Editor eine schnelle Einheit, so daß der Programmentwicklungsaufwand minimal ist. Übersetzte Programme laufen ohne weitere Hilfsprogramme auf jedem C64, nutzen den gesamten Programmspeicher des C64 und sind 3–4mal schneller als vergleichbare Programme in BASIC.

Aus dem Inhalt:
- leistungsfähiger Compiler mit Editor auf Diskette
- vollständiger Einführungskurs in Pascal
- Beispiele und Aufgaben
- Tips&Tricks für den Profi
- ausführliche Bedienungsanleitung

Hardware-Anforderung:
C64 mit Floppy 1541-/1570-/1571-Laufwerk oder C128 (im 64er-Modus) mit Floppy 1541-/1570-/1571-Laufwerk.

ISB N 3-89090-222-7

Markt&Technik



DM 52,-
sFr 47,80
öS 405,60